DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A HIERARCHY OF KNOWLEDGE LEVELS IMPLEMENTED IN A RULE-BASED PRODUCTION SYSTEM TO CALCULATE BOUNDS ON THE SIZE OF INTERSECTIONS AND UNIONS OF SIMPLE SETS

by

Barry M. Tilden

December, 1984

Thesis Advisor: Neil C. Rowe

Approved for public release; distribution is unlimited

T224414

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
A Hierarchy of Knowledge Levels Implemented in a Rule-Based Production System to Calculate Bounds on the Size of Intersection and Unions of Simple Sets

5. TYPE OF REPORT & PERIOD COVERED
Master's Thesis
December, 1984

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Barry M. Tilden

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93943

12. REPORT DATE
December, 1984

13. NUMBER OF PAGES
143

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
database abstract, bounds on statistical queries, Prolog rule-based production system, granularity of database abstract

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In this paper, the work done by Rowe in reference 1 and 2 is combined in the implementation of a rule-based system to produce upper and lower bounds and estimates as to the size of intersections and unions of simple sets. The system constructed for this paper uses the hierarchy of knowledge levels of reference 2 as the tabulated statistics in the database abstract. The system is tested and analyzed to determine the advantages and

disadvantages of increasing the knowledge level of the database abstract used for the calculation and of varying the number of partitions used in constructing the database abstracts. The paper also discusses the implementation details encountered during construction of this system in the Prolog programming language.

Approved for public release; distribution is unlimited

A Hierarchy of Knowledge Levels Implemented in a Rule-Based Production System to Calculate Bounds on the Size of Intersections and Unions of Simple Sets

by

Barry M. Tilden
Lieutenant, United States Navy
B.S., United States Naval Academy, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December, 1984

ABSTRACT

In this paper, the work done by Rowe in [Ref. 1] and [Ref. 2] is combined in the implementation of a rule based system to produce upper and lower bounds and estimates as to the size of intersections and unions of simple sets. The system constructed for this paper uses the hierarchy of knowledge levels of [Ref. 2] as the tabulated statistics in the database abstract. The system is tested and analyzed to determine the advantages and disadvantages of increasing the knowledge level of the database abstract used for the calculation and of varying the number of partitions used in constructing the database abstracts. The paper also discusses the implementation details encountered during the construction of this system in the PROLOG programming language.

TABLE OF CONTENTS

I. BACKGROUND AND THEORY 10

A. MOTIVATION FOR DEVELOPING A NEW QUERY METHOD 10

B. DEFINITION OF A DATABASE ABSTRACT 11

C. THE INFORMATION CONTENT OF THE DATABASE ABSTRACT 14

D. HYPOTHESES TO BE CONSIDERED 17

II. IMPLEMENTATION OF THE DATABASE ABSTRACT QUERY ESTIMATION SYSTEM 19

A. GENERAL INFORMATION 19

B. THE DATABASE ABSTRACT GENERATION SUBSYSTEM . . 20

1. General Characteristics of the Subsystem 20

2. The User Interface 21

3. The Partitioning Process 22

4. Generation of the Actual Database Abstracts 25

5. Processing an Example Tuple 27

C. THE QUERY SUBSYSTEM. 30

1. Structure of the Subsystem 30

2. The User Interface and Help Features . . . 33

3. The Preprocessor. 35

4. Testing for Disjoint Sets and Subsets . . 39

D. IMPLEMENTATION OF RULES FOR THE VARIOUS LEVELS 41

E. SUMMARY 49

III. ANALYSIS OF SYSTEM TESTS 50

A. TESTING STRATEGY 50
B. LEVEL OF KNOWLEDGE OF THE DATABASE ABSTRACT 52
C. GRANULARITY OF THE DATABASE ABSTRACT 56
1. Varying Granularity of All Attributes . . 56
2. Varying Granularity of Only One or Two Attributes 60
D. USE OF ALIASES TO IMPROVE SYSTEM ACCURACY AND RESPONSE TIME 73
E. CONCLUSIONS 74
F. EFFECTS OF USING PROLOG TO IMPLEMENT THE SYSTEM 80

IV. POSSIBLE EXTENSIONS/IMPROVEMENTS TO THE SYSTEM . . 85
A. EXTENDING THE SYSTEM TO ACCEPT QUERIES ON OTHER STATISTICS 85
B. IMPROVEMENTS IN SYSTEM ACCURACY 87
C. IMPROVEMENTS IN SYSTEM EFFICIENCY 89
D. SUMMARY 90

APPENDIX A: SOURCE LISTING FOR THE DATABASE ABSTRACT GENERATION SUBSYSTEM 91

APPENDIX B: SOURCE LISTING FOR THE QUERY ESTIMATION SUBSYSTEM 110

LIST OF REFERENCES 141

BIBLIOGRAPHY 142

INITIAL DISTRIBUTION LIST 143

LIST OF TABLES

I. Ten Queries Used to Test the System 51
II. Breakdown of Various Mixes of DBAs 63

LIST OF FIGURES

2.1 PROLOG Code for the Level Five Upper Bound Calculation for Intersection Sets 47
3.1 Accuracy and Speed of Response for Five Levels of Knowledge 53
3.2 Storage and Response Times for the Various Level DBAs of the Medical Database 55
3.3 Space Versus Time Plot for Various Granularities 57
3.4 Accuracy for Various Granularities 59
3.5 Response Times for Different Granularities 61
3.6 Storage Requirements of Medical Information DBAs for Various Granularities 62
3.7 Accuracy of Query 4 Varying the Granularity of DBAs 65
3.8 Accuracy of Query 8 Varying the Granularity of DBAs 66
3.9 Accuracy of Query 9 Varying the Granularity of DBAs 67
3.10 Accuracy for Two DBAs Varying the Attribute Range of the input Queries 68
3.11 Storage Requirements for DBA's of Various Granularities for the Medical Database 72
3.12 Accuracy Results for Query 4 Using the Specialized DBA 75
3.13 Accuracy Results for Query 6 Using the Specialized DBA 76
3.14 Response Times for Query 4 For the Specialized DBA 77

3.15 Response Times for Query 6 for the Specialized DBA 78

I. BACKGROUND AND THEORY

A. MOTIVATION FOR DEVELOPING A NEW QUERY METHOD

Traditional sequential methods of dealing with statistical queries on numerical databases involve the inspection of each data element within the database to determine if that element satisfies the queried condition. For a large database of thousands of items, such a search results in poor response time and tremendous use of machine resources regardless of the size of the queried set with respect to the size of the database. Even when indexing techniques are employed to provide more direct access to indexed data items within the database, query response time can be unsatisfactorily slow when the database is large and is located on numerous pages in a virtual paging system. The primary reason for this slow response is that if the database is randomly distributed on the queried attribute the data items required by the query will be dispersed throughout the database and many of the pages containing the database will have to be transferred into primary storage to retrieve the data. For databases which are extremely large, such as the U. S. Census Bureau database, such a delay can be extremely long and quite unsatisfactory. One solution to this problem, suggested in [Ref. 1], involved the use of an additional file of information about the database called the database abstract. Indeed, Rowe also suggested that access to the database itself might even be unnecessary if sufficient information could be stored in the database abstract to answer the user's queries within a satisfactory tolerance.

B. DEFINITION OF A DATABASE ABSTRACT

The concept of a database abstract should not be that unfamiliar to the reader as it is quite similar to that of a mathematical equivalence class. Recall that in mathematics when an equivalence relation is defined over a domain, all elements which are equivalent to each other can be grouped into a set called an equivalence class. If the original domain were a group (in the mathematical sense) closed under a certain operation, then the collection of equivalence classes defined by the given equivalence relation will also be a group (called the quotient group), when a new operation is appropriately defined in terms of the original operation. In the context of a database, the equivalence relation appears in the form of a partition of the database on one of its attributes. All data items which are within a particular partition with respect to the partitioning attribute are considered to be equivalent and thus the database abstract is similar to a 'quotient database'. As such the database abstract contains information about the equivalence classes rather than the individual data items. The most difficult question resulting from this approach is the problem of how to capture as much of the information contained in the individual data items of the equivalence class as possible, in the most compact storage area possible, when tabulating the information to be recorded about the equivalence class.

For the purposes of this paper, we will consider a sample database of medical information about patients in a particular hospital. Each data item consists of a six-tuple containing the patient number, sex, disease activity level, temperature, cholesterol level and prednisone level of a particular patient. Thus each data element in this database contains six attributes. If we define a partition of the

database on the sex attribute, we obtain two equivalence classes, one containing all tuples with male for the value of the sex attribute and the other all tuples with female for the value of the sex attribute. Our new database abstract contains only two data items. Obviously we cannot record a disease activity, temperature, patient number, etc for the equivalence classes because they are a different type of entity than the tuples of which they are composed. We can, however, record the mean value (just as an example) for disease activity, temperature, etc. for all the tuples in each equivalence class and still retain some portion of the information which is contained in the individual tuples.

This is the basic idea of the database abstract except that the database abstract utilizes multiple equivalence relations to formulate orthogonal equivalence classes over each of the attributes in the relation. For example, in addition to the partition discussed above over the sex attribute, the database can be partitioned over the temperature attribute. All patients with temperatures less than 35.5 are considered to have low temperatures, those between 35.5 and 36.5 inclusive to have normal temperatures and those above 36.5 to have high temperatures. Under this equivalence relation, the database is partitioned into three parts. All tuples in each partition are equivalent with respect to this relation. We can record statistical information about all the equivalent tuples in each partition and attempt to capture as much information about these tuples as possible. In like manner the database can be partitioned over each of its attributes whether they be numerical or non-numerical and thus the final database abstract can be thought of as an orthogonal collection of equivalence classes with certain selected statistical information recorded about each equivalence class in an attempt to capture many of the characteristics of the database in a

much smaller storage area than the actual database would contain.

Even though attributes may be partitioned into equivalence classes regardless of whether they are numerical or non-numerical, there is an inherent difference in the flexibility of the partitioning process between these two types of attributes. For instance, in our example above concerning the temperature attribute, we may just as easily have decided to divide the database into low temperatures, low normal, high normal and high temperatures instead of the three partitions defined above. This new partitioning would simply involve a redefinition of the boundaries of the partitions. In fact, the database could be divided into any number of partitions between two and the number of distinct values for temperature in the database depending upon the desires of the user. Now consider the non-numeric attribute, sex. It must be partitioned one and only one way because it has only two distinct values. Therefore during this partitioning process, we must handle numeric and non-numeric attributes in a slightly different manner.

The above discussion has raised two important questions which we must consider in constructing the database abstract. First, for the numeric attributes, how many partitions should be chosen for each attribute to maximize the amount of information about the original database which is captured in the database abstract while still minimizing the storage space required to store the database abstract. This question will be analyzed in considerable depth in chapter three. The second question concerns what information should be recorded in the database abstract about each equivalence class in crder to again maximize the real information content while still keeping the size of the database abstract as small as possible.

C. THE INFORMATION CONTENT OF THE DATABASE ABSTRACT

In [Ref. 1], that author included 16 statistics for each equivalence class in his database abstract in order to be able to service a large number of different queries on various statistics. In this discussion, we restrict ourselves to queries on the size statistic in order to more easily direct our attention to other questions about the formulation of the database abstract. Here the size statistic refers to the number of database tuples which fall into a particular set defined by the user. The barious methods to define these query sets will be discussed later in this chapter. For the information content of the equivalence classes constituting our database abstract, we have used the five levels of knowledge defined in [Ref. 2]. These levels represent increasing quantities of information and storage requirements and result in increasing accuracy of query estimation results.

As discussed in [Ref. 2], the first level of knowledge contains only the size of each equivalence class. Therefore this database abstract would contain the size of each partition of each attribute in the database. The total information content would depend on the number of partitions chosen for each of the numerical attributes in the database. The second level of knowledge contains in addition to the size information, the mode frequency and number of distinct values of each attribute in every equivalence class in the database. Therefore, if each of the five numerical attributes in our medical database were divided into four partitions and the non-numerical sex attribute into two partitions this database abstract would contain 2*(4*5*5 + 2*5) or 220 additional data items. This level of knowledge is equivalent to that referred to as level 2a in [Ref. 2]. The level three database abstract for this system

corresponds to the level 3a abstract of [Ref. 2] and includes the frequency of the second most frequent value and the frequency of the median frequency value for each attribute in each equivalence class and therefore adds an additional 220 data items to the database abstract file. Level four is a logical extension of levels two and three and contains a list of the frequency of the most frequent through the kth most frequent value for each attribute in each equivalence class in the database where k represents the number of distinct values of the given attribute in that equivalence class. Finally, the highest level of knowledge is the tagged frequency distribution list of level five. This database abstract contains a list of the values of each attribute in each equivalence class together with the frequency of occurrence of that value in the given equivalence class. This is the highest level of knowledge in the context of this hierarchy because it represents the size of all first level intersections of all sets in the database and as such corresponds to level one knowledge if we had considered these first level intersections as the primitive sets in the database.

The choice of these particular items of information is not a hard and fast requirement for a system of this type. Other items of information may be substituted for those included above. For example, instead of the second most frequent value in level three, the least frequent or anti-mode frequency of each attribute in each equivalence class could be chosen. The important requirement for the purposes of this system is that the levels of knowledge represent a sequence of increasing information content in the database so that we can demonstrate an increasing accuracy of query estimate results and weigh these benefits against a greater storage requirement and a decreasing speed of execution as higher level database abstracts and their correspondingly

more complex rules are used. In fact, other information besides frequency information could conceivably be used to capture the information content of each equivalence class, however the frequency information discussed above seems to fit more appropriately into a well defined hierarchy than any other types of information considered.

Now that the information content of the various levels of knowledge has been described, an appropriate operation on that information must be defined in order to gain the best possible estimate of the size of any queried set. Note that the basic operation on the original database in order to answer a query is the operation of search and comparison to the given input conditions. Because the database abstract contains a more abstract and concise representation of the information content of the original database, the basic search operation will no longer suffice. A new operation must be defined in terms of the base operation in a well defined manner to produce accurate estimates to user's queries in much the same manner that a new operation is defined on a quotient group in terms of the original operation. The term operation is used quite loosely in this context because it actually takes the form of a database of rules which extract the queried information from the information content of the database abstract. The search and comparison operation is used to locate and apply the appropriate rule for a given set of input conditions in the form of a query. The following chapter will describe this database abstract operation by describing the rules used in the prototype system constructed to implement such a query estimation system.

D. HYPOTHESES TO BE CONSIDERED

The remainder of this paper will describe a system which calculates upper and lower bound and estimates for the size of a given set using only the information contained in the database abstract. Various characteristics of the database abstracts will be varied and the system will be tested to determine what effect these characteristics have on the system responses. At this point it is appropriate to hypothesize about the expected effects of varying these characteristics.

First of all it is expected that using higher levels of knowledge will produce tighter upper and lower bounds but will also result in increased system response times and greater storage requirements for the database abstract. It is also suspected that there is one level which is the best to use in the sense that it causes much tighter upper and lower bounds than the next lower level but results in only a modest increase in the response time and storage required over the next lower level. Through our testing we will attempt to determine if such a 'best' level exists and if so, which of our levels it corresponds to.

The second hypothesis for the system is that as the number of partitions of the database abstract increases, the system will produce tighter upper and lower bounds but will suffer from longer response times and larger storage requirements for the database abstract. It is also believed that there exists an optimum granularity for the databse abstract in the sense that increasing the number of partitions above that granularity will result in only slightly tighter bounds but will result in constant increases in system response times and storage required for the database abstract. Through our testing we will attempt to determine what this optimum granularity is for a sample database and

what characteristics of that database determined this optimum granularity.

The third and final hypothesis for this system concerns the method that the user uses to input his query set. In order to make the system as general purpose as possible the user is allowed to enter his sets by specifying ranges of attributes. For example, using the sample medical database utilized throughout this paper, if the user wishes to know the size of the set of all patient with temperatures between 36 and 37 who also have a cholesterol level between 230 and 250 he may specify the query set:

temp(36,37) * chol(230,250).

This method has the advantage of being very general purpose but suffers the penalty of less accurate bounds generated by the system as will be seen in chapter three. If the user queries the actual partition sets of the database abstract, the bounds produced by the system will be much tighter, however, the user may have no need to know the size of such sets. As a solution to this conflict between tighter bounds versus a more general query method, we submit that for many databases, there exist predefined ranges of attribute values which are queried often by the user community. If such ranges were forced to be actual partition sets during the generation of the database abstract, a significant improvement in the tightness of calculated bounds could be achieved when such ranges were queried by the user. Testing on such a specially constructed database abstract will be conducted to determine if the expected benefits are in fact as great as expected.

II. IMPLEMENTATION OF THE DATABASE ABSTRACT QUERY ESTIMATION SYSTEM

A. GENERAL INFORMATION

The previous chapter introduced the operation on the database abstract equivalence classes as a database of rules which extract the queried information from the primitive information stored in the database abstract. This database of rules implies that the most natural method to implement such a system would be as a rule-based production system where the system must decide which rules apply to the given query and apply only those rules to produce the most accurate result. Such a prototype rule-based system was implemented as a part of this project again utilizing the medical database described in chapter one. This chapter will describe in some detail the structure and organization of this system (to be called DAQUES, for Database Abstract QUery Estimation System for the remainder of this paper).

The first major decision to be considered before commencing the construction of this system was the programming language to utilize. After considering both Lisp and PROLOG, the advantage of faster speed of execution because of the Lisp compiler seemed tc be surpassed by the more natural syntax and excellent pattern matching capabilities of PRCLOG. Because this system is only a prototype system, speed of execution was not considered to be as important as the ease of understanding the logic of the rules in the rule base provided by the PROLOG language. Additionally, this author also believes that the power of logic programming is just now being recognized and such languages may very well become the principal ones for artificial intelligence

programming applications in the very near future. It seems advantageous for computer scientists outside of Japan to begin to learn the strengths and weaknesses of this language.

The DAQUES system consists of two major parts, the first being the database abstract generation portion and the second the actual query system. These subsystems are almost completely independent of each other, their only link being that the output of the generation subsystem is the input to the query subsystem and the file 'attrlist' which is consulted by both subsystems. The subsystems will be described in the following two sections.

B. THE DATABASE ABSTRACT GENERATION SUBSYSTEM

1. General Characteristics of the Subsystem

The database abstract generation subsystem is not intended to be used by the average database user with only query privileges. Use of this system will require a relatively sophisticated knowledge of the meaning of the term, database abstract, and of the the merits of the various tradeoffs which must be made when constructing the database abstract. For this reason, it is recommended that the use of the database abstract generation subsystem of DAQUES require the user to hold the database administrator privilege or its equivalent.

The database abstract generation subsystem is constructed to be as general as possible so that it can be applied to an arbitrary database. However, there are a minimal number of facts in the system which are specific to the example medical database. These facts are located at the end of the 'attrlist' file and provide a list of the attibute names for the database, a designation of the attributes which are non-numeric and their values in the

database. These data items would be the only portion of the entire system which would have to be altered to port the system to another database.

2. The User Interface

The most visible portion of the database abstract generation subsystem is the user interface section which prompts the database administrator for all required input information needed to construct the desired database abstract and converts that information into a form which the rest of the program can understand. The information needed is the name of the database to be processed, the level of knowledge for the particular database abstract to be constructed (could be 'all' if the administrator desires to construct four levels at once), the number of partitions that each attribute within the database is to be divided into and the name of the output file or files in which to store the database abstract. The name of the database file may seem to be unnecessary information, after considering that several rules in the system are specific to a particular database. With the current system it is true that the database name is unnecessary. However, it is possible to process one or more databases with no attribute names in common and therefore include all necessary rules for all of the databases. The rules would not interfere with each other because the clause heads of the rules for the inapplicable database would not match, given the distinct attribute names. Therefore, to provide added flexibility, this information is included as part of the input from the database administrator.

The level of knowledge to be input must be one of the five discussed in chapter one or the user may specify 'all' levels. The 'all' option is a bit of a misnomer because it calculates only levels two through five. This

distinction is because of the fundamental difference between the calculations for level one and those for levels two through five to be discussed later in this chapter. The advantage to choosing the 'all' option is that the database need only be processed once to produce four different levels of knowledge whereas, if a particular level is specified, the database is processed once for each level. Therefore, the 'all' option can provide a significant savings in time and machine resources especially for a large database. If the user specifies the 'all' option, he must also specify a list of output files in which to store the database abstract files for levels two through five instead of a single file name.

3. The Partitioning Process

The most difficult and important decision which the database administrator must make concerns the granularity of the partitions for each attribute. If the attribute is non_numeric, there is only one acceptable number of partitions and it is input from the database specific facts in the 'attrlist' file. If the attribute is numeric, the user must input the desired number of partitions for that attribute. The attribute list file is then checked to see if the partition boundaries have previously been calculated for this granularity. If so, the tabulated boundaries are used. If not, the user is asked to choose between automatically generated partition boundaries or specific boundaries which he may input himself.

The former method for determining the partition boundaries uses an algorithm to attempt to place an equal number of values in each partition. The input to this rule is the number of partitions, and a list of the values of that attribute together with their frequency of occurrence in the entire database. This list is equivalent to level

five knowledge about the entire database and must be produced by the database administrator prior to utilizing the system. This information is easily generated by running this same system and specifying 'one' as the number of partitions. The algorithm divides the number of values in the database by the granularity and then passes through the frequency list including each value until its inclusion will cause the number of values in the partition to exceed the calculated size of the partition. The actual size of the first partition is then subtracted from the size of the database. That number is divided by the number of partitions minus one and the algorithm is called recursively. When the database contains a large number of distinct items, none of which occur very frequently, (i.e. even distribution), this algorithm is quite good for producing partition boundaries resulting in partitions of relatively equal size. However, when an attribute has one or more values which occur many more times than the other values in the database, (i.e. the distribution is highly uneven), this algorithm results in partitions of very different sizes and consequently, as will be evident in our discussion of the query subsystem, reduced accuracy of query results.

In the case of the highly uneven distribution, or to create partition sets which are meaningful in a particular application field, the administrator may explicitly specify the boundaries of the partitions. This option is a highly desirable one especially in the latter circumstance. Many attributes fall naturally into partitions because of conventions within the application area or merely for convenience. As mentioned in chapter one during the introduction to the meaning of the database abstract partitions, the temperature attribute may be divided into low temperatures, normal temperatures and high temperatures by specifying the numeric convention for defining these terms. In the context of our

program, such user specified partition boundaries may not result in partitions of relatively equal size, however the partitions themselves will be sets which are much more meaningful to the expected user of the system than the algoritmically produced partitions. For reasons that will be discussed in depth later in this chapter, there are also tremendous advantages in accuracy of the bounds calculations for the system user when his query set corresponds exactly to a union or intersection of the exact partition sets rather than numerically specified boundaries.

For example, if the user is interested in the set of male patients with temperatures between 35.5 and 36.5, he may explicitly query this set with no knowledge of the partitions contained in the database abstract. If this set does not correspond exactly to the boundaries generated by the partitioning algorithm, the query preprocessor must determine the minimal union of actual partitions which covers this desired set and calculate the upper bound for the query based on this set. In a similar manner, to calculate a lower bound on the query, the query preprocessor determines the maximal union of actual partition sets which is completely included in this desired set and calculates the lower bound of the query based on this set. If, on the other hand, the database administrator is aware of commonly queried sets in the application area such as the normal temperature range, he may explicitly specify that the partition boundaries correspond to these ranges, providing much more accurate query results for such sets.

To extend this concept one step further, the database administrator may specify an alias for that partition name such as 'normtemp'. Using this facility allows the user to express his query in terms of the 'normtemp' set instead of the conventional name for the second partition set for the temperature attribute, i.e. temp(2). The

advantages of explicitly specifying partition boundaries for particular attributes should become more apparent after reading the discussion of the query subsystem of DAQUES.

Whichever method the database administrator chooses to arrive at the boundaries of the partitions, these boundaries are inserted into an attribute list and added to the 'attrlist' file for use in later database abstract generation executions and for use by the preprocessor of the query subsystem. Additionally a list of the attribute names and the number of partitions is written into the level one database abstract file and is passed on to the 'frequencies' file which performs the actual database abstract generation function.

4. Generation of the Actual Database Abstracts

First we will discuss the level one calculation. As mentioned previously, there is an inherent difference between the method of calculation for the level one database abstract file and the remaining levels. Recall that level one information contains the size of each partition with respect to each attribute in the database. When the user interface rules determine that level one knowledge is desired by the database administrator the 'level1' file is consulted. Consequently, when the 'ex' function is called by the user interface, it matches the level one 'ex' rule and begins the calculation of the level one database abstract. The first task for this calculation is the initialization of the totals for the size of each partition located in the PROLOG internal database to zero. Then each tuple of the database is read in from the database file. The 'check' function is called for each of the attributes in the list passed to this file by the user interface. The 'check' rule determines the value of the given attribute in the current tuple. It determines the partition set into

which this value falls and increments the total for that partition set in the internal database by one. After the entire database has been processed in this manner, level one information is contained in the PROLOG internal database. The 'continueall' clause accesses all of this information and writes it to the external file specified for the level one database abstract by the database administrator.

When the database administrator specifies any level other than level one, the user interface rules consult the 'frequencies' file as well as the corresponding 'level' file. The 'frequencies' file produces the same information for levels two through five and asserts that information into the PROLOG internal database for use by the appropriate 'level' file. The information produced by the 'frequencies' file is exactly the level five database abstract information and from this information, levels two through four information can be derived. Recall that level five information is a list of the values that occur for each attribute in each of the partition sets together with the frequency with which those values occurred in that set. Therefore the first task accomplished by the 'frequencies' file is to assert an empty list as the initial tagged frequency distribution list for each attribute and each partition set in the database abstract. Then, each tuple is read from the database file. The 'checkallattrs' rule is invoked for each attribute. This rule finds the value of the 'partitioning attribute' from the current tuple and determines which partition set that tuple belongs to with respect to the 'partitioning attribute'. Then the 'check' rule is invoked for each numeric attribute. This rule finds the value of the 'check attribute' and accesses the frequency list for the partition set of the 'partitioning attribute' with respect to the 'check attribute'. The 'intolist' rule searches the list for the value of the 'check attribute' and increments the

frequency corresponding to that value by one if the value is found. If the value is not found in the list, the value is inserted into the list with a frequency of one, maintaining the ascending order of values (not frequencies) within the list. This process is repeated with each numeric attribute substituted as the 'check attribute'. Then the 'checkallattrs' rule is called with the next attribute substituted as the 'partitioning attribute'.

5. Processing an Example Tuple

As an example of this process, recall the medical database with its six attributes; patient number (patno), sex, disease activity (da), temperature (temp), cholesterol level (chol), and prednisone level (pred). The values of these tuples are arranged in that order within the input tuples so that an input tuple would appear as follows:

(10,female,12,36.2,230,55)

The 'checkallattrs' rule would first be invoked with the patno attribute as the 'partitioning attribute'. The value of 10 for patno would place this tuple in the patno(1) partition, for example. Then the check rule would be invoked with the patno attribute as the 'check attribute' and the following list would be searched for the value 10:

data(freqdist,patno(1),patno,[[4,1],[7,2],[10,3],[15,6]])

In this case, 10 would be found and its frequency, 3, would be changed to 4. Then the check rule would be invoked with da (the next numeric attribute) as the 'check attribute' and the following list would be searched for the disease activity value of 12:

data(freqdist,patno(1),da,[[5,3],[14,4],[24,5]])

In this case the value of 12 would not be found so the two element list [12,1] would be inserted into the list to produce the following new list:

data(freqdist,patno(1),da,[[5,3],[12,1],[14,4],[24,5]])

In a similar manner the 'check' rule would be invoked with the attributes temp, chol and pred as the 'check attributes'. Then with the patno attribute execution of the 'checkallattrs' rule completed, this rule is invoked with the sex attribute as the 'partitioning attribute'. The partition set 'female' is determined to be the one to which this tuple belongs with respect to the sex attribute and the 'check' rule is invoked with each numeric attribute as the 'check attribute'. In like manner, the 'checkallattrs' rule is invoked with the da, temp, chol and pred attributes as the 'partitioning attribute' to complete the processing of this one tuple. Then the next tuple is read in and the process repeated.

When the entire database has been read and processed, the internal PROLOG database contains tagged frequency distribution lists for every partition and each attribute. A good way to visualize this information is to recall that the partitions or equivalence classes of our database abstract are the primitive data items of the 'quotient database'. We are attempting to capture as much of the information content of the tuples of the original database as possible in our database abstract. We cannot tabulate a value for each of the attributes for the partitions because the partitions are made of a collection of tuples of the original database. Instead we chose in level five to substitute this tagged frequency list for each attribute as the 'abstract value' of that attribute. In terms of our example, instead of tabulating a da value for the patno(1) partition we tabulate the list of the da values from all the

tuples which fall into the patno(1) partition and the frequency of occurrence of each value in that partition.

Now that all this information has been asserted into the PROLOG internal database, the 'frequency' file invokes the 'continue' rule. The content of this rule will depend on which 'level' file was consulted by the user interface after inputting the desired level of knowledge from the database administrator. If the desired level is five then the 'continue' rule merely writes all of the tagged frequency distribution lists to the external file designated by the database administrator to receive the level five database abstract. If level four is chosen, the 'continue' rule compacts the tagged frequency lists removing the values of the attributes and leaving only the frequencies. These frequencies are then sorted in descending order and written to the file specified for the level four database abstract. If level three is designated, the tagged frequency list is sorted in descending order of the frequencies and the second and middle elements of the list are extracted, corresponding to the frequencies of the second most frequent and the median frequency value of the check attribute on that partition. These values are written in an appropriate format to the file designated for the level three database abstract. Finally, if level two is specified, the tagged frequency lists are again sorted in descending order of frequencies but the first frequency in each list and the length of the list is written to the appropriate file, representing the frequency of the most frequent value and the number of distinct items in the partition.

It should now be obvious to the reader that the calculation of the database abstracts for levels two through five involves a common initial calculation. When the database administrator specifies 'all' as the level of knowledge for the database abstract calculation, the 'alllevels' file

is consulted in place of a specific 'level' file. The 'continue' rule in this file invokes four individual rules called 'continue2', 'continue3', 'continue4', and 'continue5', all identical to the 'continue' rules contained in their respective 'level' files. In this way the 'continue' rule for the 'alllevels' file processes the tagged frequency distribution lists located in the PROLOG internal database producing all four desired database abstract files.

Now that the database abstract generation subsystem of the DAQUES system has been described in detail, we will continue our discussion in the following section with a description of how these database abstract files are used by the query subsystem to produce bounds and estimates for the user's queries.

C. THE QUERY SUBSYSTEM.

1. Structure of the Subsystem

After the database administrator has completed the construction of the database abstract files, the second subsystem of the DAQUES system should be ready to be used with no modifications. This subsystem was designed to input all database specific information, such as rules concerning attribute names, from the database abstract files and the 'attrlist' file. Therefore, the query subsystem should be able to query multiple database abstracts independent of their particular partitions or even the database from which they were constructed, provided that the database abstract was built entirely with the generation subsystem of the DAQUES system. Of course, this system is designed for use by the standard user with only query privileges.

The query subsystem has three major program parts, the user interface and help feature, the query preprocessor and the actual calculation portion of the program. With

regard to file organization, the rules are divided into five files. The 'master' file contains the user interface and all preprocessing functions as well as many rules too general to be included in one of the specific files (e.g. the 'member' rule). The 'info' file contains the rules for the help feature of the user interface. It is currently a rather limited feature but was written with future expansion in mind. The 'sups' file contains all the rules for calculating the upper bound on the size of a query. It has its rules divided into those for unions and those for intersections and then subdivided into rules for levels one through five within the union and intersection divisions. The 'infs' file contains the rules for calculating the lower bound on the size of the queried set. Its organization is identical to the 'sups' file. Finally, the 'ests' file has all the rules for calculating an estimate for the size of the input set. It has only distinct sections for union and intersection rules.

Within the 'sups', 'infs', and 'ests' files the basic program structure is quite similar. The 'dosup', 'doinf', and 'doest' rules each input a statistic, a set, and an attribute. The statistic will always be 'size' in the current system implementation, however, the system is designed so that additional rules might be added at a later time to extend the capabilities of the basic system. Also for this reason the attribute is input to these rules to allow future queries such as the mean value of temperature for the input intersection or union set. With this extensible format, a general purpose user interface and preprocessor could invoke the appropriate 'do' rule merely by passing the proper arguments to the rule.

When the query sets are input to the 'do' rules, they are in prefix form or in PROLOG terminology, they consist of a single argument functor, where the functor name

must be either 'and' or 'or'. The single argument to these functors contains a list of the sets in the intersection or union and as such can be of arbitrary length. Additionally any individual element of the list can be one of these single argument functors indicating a lower level union or intersection. Only time constraints limit the size of the intersection or union list or the nesting level of unions and intersections within unions or intersections. Because of this PROLOG list structure embedded within a functor, the system becomes very general purpose and handles a very wide assortment of input query sets. Also such questions as the optimum normal form of an input query become much less important than they are in systems with a much more structured input set. In [Ref. 1], for example, the query system inputs sets in prefix form with two arguments only. Therefore, in order to query a set such as and(lowtemp,lowchol,male), that system had to decide which of the three possible formats, and(and(lowtemp, lowchol), male), and(lowtemp, and(lowchol, male)) or and(and(lowtemp, male), lowchol), produced the best query results. (In his system, Rowe decided to process all three query forms and use the most accurate results.) Such questions do not apply in a list oriented input format system.

The list format for the input sets does require a very different processing strategy, especially when the next set in the input list might very well be a very large union or intersection of other sets. Tests such as disjoint and subset rules become much more complicated because they must be more general purpose than systems with a fixed number of arguments. The general processing strategy for this format involves examining each set in the argument list. The set is checked to see if it is a 'simple' set in the sense that it has size information stored about it in the PROLOG internal database or in the system cache. If it does, the

program retrieves this information and moves on to the next set. If the current set has nc data either in the internal database or in the system cache, then the set is not 'simple' and the appropriate 'do' rule is invoked for this set. When this 'do' rule returns from execution, the set will have been converted to a 'simple' set in that it will have all the same information stored about it in the system cache that an actual partition set has stored in the internal database at the time program execution started. Another important point is that if this same union or intersection occurs at another point in the same query or in a subsequent query within the same session, the union or intersection set still appears as a 'simple' set to the system. A more thorough discussion of the details of this concept will be presented following the presentation of the user interface and preprocessing sections of the query subsystem.

2. The User Interface and Help Features

Because this system is intended to be used by the novice user, it includes a very friendly user interface and an expandable help facility. The user interface must input several items of information from the user and using this information, must control the ccnsulting of the proper files to ensure the results desired by the user. The user interface first asks the user what level of knowledge he wishes to use for his query and gives the user the choice of a normal, alternate, or special database abstract file. (It is intended that database abstracts with several different granularities of partitions or aliases will be available to the user depending on his accuracy needs or the partition conventions which he commonly uses.) If the user does not understand the different levels of knowledge or does not remember which database abstract files are currently associated with the terms normal, alternate and special, he may

simply respond by typing 'help'. This will produce an explanation of the current database abstracts available as well as a discussion of the differences in levels one though five knowledge. After the user has entered this information, the user interface must consult the appropriate database abstract files as well as the alias file and attribute list files discussed in the section on the database abstract generation system. The user interface then stores knowledge of the current level that the user is utilizing so that during the next query, no additional files need be consulted unless the user wants to use a new level of knowledge or a new database abstract file.

The user interface next inputs the statistic to be queried which currently must be the 'size' statistic. If the user inputs a different statistic, the help feature will provide an explanation that only the 'size' statistic is currently operational on this system. It is anticipated that when other statistics such as mean are provided in the system, the user interface will next ask the attribute that the statistic is to be queried over. Because the size of a set is not associated with any particular attribute, the system currently omits this question.

The user interface next inputs the set to be queried in infix form. Again failure to input a proper set will result in a call to the help facility and an explanation of the proper format for inputting sets. It was decided to input the query set in infix form after several weeks of testing the program using the prefix form as input for the sets. The prefix, list oriented set notation is quite useful for providing generality to the program, however, requiring the user to happen upon the proper combination of parentheses and square brackets needed to input the desired set completely counteracted all other attempts made to make this interface user friendly. As an example, consider the

query wishing to know the size of the set of all males in either the first or second cholesterol partitions. The prefix form of this sets is the following:

```
and([ male,or([chol(1),chol(2) ]) ])
```

Compare the above notation with the infix notation shown below:

```
male * (chol(1) + chol(2))
```

It should be obvious how this advantage in simplicity becomes increasingly more important as query sets become more complicated. Therefore it was decided that the small degradation in system performance needed to implement an infix to prefix preprocessor was justified. This preprocessing feature is described in the following subsection.

3. The Preprocessor.

One need for a set format preprocessor was introduced in the previous section. This subsection will demonstrate two additional needs for preprocessing and describe the specific performance of the rules to implement these functions.

During the first subsection above the general nature of the input set was discussed in some detail. One of the disadvantages of this format is the capability it provides to the user to input sets which execute much less efficiently than a different expression of the same set would execute. For example, the set:

```
chol(1) + chol(2) + chol(3)
```

translates into the prefix form:

```
or([chol(1),chol(2),chol(3) ])
```

which executes much more efficiently and accurately than the query:

(chol(1) + chol(2)) + chol(3)

which translates into prefix as follows:

or([or([chol(1),chol(2)]),chol(3)]

The two sets are logically identical and should result in the same upper and lower bounds and estimates from the system. However, they would not without the second function of the preprocessor. This function is descriptively called the 'squash' function because of its capability to transform a multilevel set into a logically equivalent single level set which will produce quicker and more accurate results. (For a thorough discussion of the 'optimal form' of a query, refer to [Ref. 2]). This function does not just provide protection against the novice user who inserts extra parentheses where they are not needed. The output of the third feature of the preprocessor, described below, is often in a form which needs to be 'squashed' to execute efficiently and accurately.

During the discussion of the alias structure of the generation subsystem, it was mentioned that, in general, the input sets to the system would be in the form of unions or intersections of sets defined by a range of values of a particular attribute. So far we have only discussed sets which are exactly one partition set of a particular attribute. It was also mentioned that such exact partition sets, especially when they are aliased to a descriptive name, are quite useful when a particular attribute was partitioned during the database abstract generation, to correspond exactly to a set which is of some importance in the application field and therefore would commonly be a part of various queries. However, when the above conditions do not apply,

it is appropriate to allow the user to express the query set in terms of a range of values over a particular attribute. The query subsystem allows the user to express his query set in either of the formats as well as in terms of some alias name for a particular partition set. The third feature of the preprocessor provides the conversion of these forms to a common form to pass on to the calculation section of this subsystem.

To provide a better understanding of the three stages of the preprocessor, we will follow the execution of a particular query set from its entry into the system until it is passed to the calculation system for actual processing. Consider the following query:

male * chol(115,280)

which means that the user wishes to know the size of the set of all male patients with cholesterol levels from 115 to 280 inclusive. The user interface passes this infix form to the 'convert' rule which matches the '*' symbol producing an 'and' functor with arguments of whatever is returned from the recursive calls to 'convert' made for the left and right sides of the '*' symbol. Had either of these sets been a lower level (parenthesized) union or intersection, then one of the elements of the functor's argument list would have been another functor. In fact, for infix expression without parentheses that are longer than two elements, the 'convert' rule uses left precedence rules and operates as if there were parentheses around the righthand side of the expression producing an output prefix expression of depth one less than number of terms in the the input expression. This limitation is solved by the 'squash' function.

When the 'convert' rule is invoked for a simple set like 'male', it recognizes this set as a simple set and merely returns the set name. When it is invoked with an

alias name, it checks the alias list and returns the name of the partition set to which the alias refers. If the rule is invoked with a range expression such as chol(115,280), it recognizes the two arguments to chol as lower and upper limits to a range and calls the 'cover' and 'iscovered' rules. The 'cover' rule uses the upper and lower limits and the partition boundaries stored in the 'attributelist' facts to find the minimal union of actual partition sets for the given attribute which covers the desired range. It returns this union as the set to be used by the calculation system when the upper bound or sup of the statistic is being determined. Then the 'iscovered' rule uses the same information to find the maximal union of actual partition sets for that attribute which is covered by the given range. This union is used by the 'infs' file when calculating the lower bound on the size of the set. Suppose the 'cover' and 'iscovered' rules return or([chol(1), chol(2)]) and or([chol(1)]) as the two unions discussed above. Then the two sets to be passed on to the 'squash' function for the next stage of preprocessing would be and([male, or([chol(1), chol(2)])]) and and([male, or([chol(1)])]). When 'squash' is called with the first set description it returns the same form because there is no 'squashing' possible on this set. Consider, however, if the input set had been a union instead of an intersection. Then the input of the squash function would be or([male, or([chol(1), chol(2)])]) which would be 'squashed' into the more accurate and efficient form or([male, chol(1), chol(2)]). For the second set the 'squash' function recognizes the single element union which results from the generality of the 'iscovered' rule. This set will therefore be 'squashed' into and([male, chol(1)]), a form which will provide much more accurate and timely results.

Now that we have described how the user's query is preprocessed from its various possible input formats into a common form, in the following subsections we will describe how the actual calculation rules are implemented to produce the query results.

4. Testing for Disjoint Sets and Subsets

The detection of sets in the input union or intersection which are disjoint or which are subsets of each other can greatly increase the accuracy of the query results. For this reason, the calculation rules contain complex tests for these conditions. The disjoint test primarily involves a search for sets which represent two different partitions of the same attribute which, by definition, are disjoint. In addition, the disjoint test involves a complex series of tests to determine whether a union or intersection of sets is disjoint from another union, intersection or simple set. The rules which are implemented in this section are discussed in detail in [Ref. 1].

The advantage of determining that an intersection contains two or more sets which are disjoint is that all further calculations may be skipped because the size of that intersection is most certainly zero. However, the advantage of determining that two or more sets in a union are disjoint is not so straightforward. If all the sets in the union were found to be disjoint from all others in the union, then we could arrive at the size of the set by merely adding all the sizes of the component sets. However if only two sets in a very large union list are determined to be disjoint, what advantage have we gained? We could consider this disjoint union as a simple set with a definite size equal to the sum of the sizes of its two components. There are two problems with this approach. The first is the question of what to do about the nontransitivity of disjointness. For

example, if set A is disjoint from set B and set B is disjoint from set C, then we cannot conclude that set A is disjoint from set C. Therefore how do we decide which of the two pairs of disjoint sets to combine into a single set? The second problem relates to how to convert this disjoint union into a true simple set in the sense that it has all the information tabulated about it in the cache that an actual partition set would have tabulated about it in the PROLOG internal database when program execution started. If the desired level of knowledge is one, there is no problem because the size of the set is level one knowledge. However, if the desired level is five, the level five information about this disjoint union would have to be calculated and added to the cache. Because of these difficult questions, it was decided not to test for disjointness during the union calculations. The level one upper bound on the size of a union assumes the worst case, that the component sets are disjoint, in arriving at the upper bound. So even for this single level where the majority of the problems discussed above do not apply, the disjoint test would not demonstrate any possible advantage.

It would be an interesting extension to this system to include a disjointness test in the union calculation, to develop rules to choose between conflicting pairs of disjoint unions and to write special rules for calculating the level two through five information about the disjoint union given only the level two through five information about the component sets. This extension to the basic system has great potential for improving system accuracy because of the prevalence of disjoint unions as input sets resulting from the preprocessor translation from range values into unions of partition sets as discussed in the previous section.

The subset test cannot generally arrive at a specific answer for the size of an input set as the disjoint test can. It can, however, shorten the union or intersection list for which the bounds and estimate calculations must be made. The 'testallu' and 'testalli' rules perform this subset test for the union and intersection rules respectively. The 'testallu' rule inputs the union list and determines whether any set in the list is a subset of any other. If so, the subset is removed from the output list because such a subset could not possibly contribute any tuples to the union which have not already been contributed by the set which contains it. In a similar manner for intersections, the 'testalli' rule discards the supersets because they cannot possibly contribute any tuples other than those contributed by the subset. The specific subset rules implemented in this system are discussed in some detail in [Ref. 1].

The disjoint and subset rules might be more appropriately included in the preprocessor section of the query subsystem because they do actually alter the input to the actual calculation portion of the system in the case of the subset rules or abort the calculation in the case of the disjoint rules. It was decided to have these rules as part of the calculation section instead because they deal with the content of the sets in the input unions and intersections. The preprocessor is mostly concerned with arriving at an optimum format with which to enter the input set and not so much with the meaning of the sets involved.

D. IMPLEMENTATION OF RULES FOR THE VARIOUS LEVELS

The actual calculation portion of the query subsystem receives three pieces of information from the preprocessor. These are the statistic being queried, the set being queried

in prefix notation, and the level of knowledge desired by the user. This information is contained in the three of the arguments of the 'dosup', 'doinf', or 'doest' rules called by the preprocessor. Depending on the value of these arguments any of several different rules could be invoked to calculate the appropriate bounds and estimates. The current system contains rules for answering queries on the size statistic only. (Chapter IV contains a discussion on how to extend the system to other statistics.) The prefix form of the set will have either an 'and' functor or an 'or' functor and will match a different set of rules in the 'sups', 'infs' and 'ests' files depending on the value of this functor in the set notation. Finally, the level of knowledge specified by the user will cause only the appropriate calculations for the desired level to be executed.

As an example of this rule matching process, we will continue the example that was started in the previous section. This section concluded that the preprocessor had converted the initial input set into and([male,or([chol(1),chol(2)])]) for the upper bound calculation and and([male,chol(1)]) for the lower bound calculation. Suppose also that the user had desired that level five be used to calculate the size of this set. In this case, the preprocessor would call the 'sup' portion of the query subsystem with the goal:

```
dosup(size,and([male,or([chol(1),chol(2)])]),_,Sup,5).
```

and the 'inf' portion of the subsystem with the goal:

```
doinf(size,and([male,chol(1)]),_,Inf,5).
```

The PROLOG system attempts to match these goals with the clause heads of all the rules in the 'sups' and 'infs' files respectively until a rule with an appropriate clause head is found. The only rule in each of these files which will

match will be the appropriate general upper and lower bound rules for calculating bounds on intersection sets thus further restricting the possible rules which could be applied. Finally, the general 'dosup' and 'doinf' rules call each of the 'dosup1' through 'dosup5' rules and 'doinf1' through 'doinf5' rules passing to these rules the set to be queried and the level of knowledge desired by the user.

The level one calculations are unique from the remaining levels in that they are always executed regardless of the specified level. Level one calculations are always executed because they are a part of the fundamental structure of the query subsystem. The level one 'dosup' rule calls the 'supall' clause passing to it the list of the component sets in the union or intersection. The 'supall' clause tests each component set to determine if that set has data stored about its size in the internal PROLOG database or in the system cache. If the set happens to be an actual partition set of the database abstract as is the case with the set 'male' above, the level one database abstract will have the actual size of the set recorded in the internal database. This size will merely be returned as an upper bound on the size of this component set. If, however, the set has no information stored about it in the internal database, as in the case of or([chol(1),chol(2)]) in the example above, the 'supall' rule recursively calls the 'dosup' rule with this component set as the argument.

The purpose of this call to the 'dosup' rule is not only to return an upper bound on the size of the component set, but to convert that set into a true 'simple set' with respect to level five information. The term 'simple set' means that the set should have all information stored about it in the system cache that an actual partition set would have stored about it when system execution begins. In the

case of our example, the set or([chol(1),chol(2)]) would have to have an upper bound on its size and an upper bound on its level five tagged frequency distribution list asserted into the system cache to fulfill these requirements. The lower level call to the 'dosup' rule executes exactly the same as the upper level call to that rule except that when the supall clause tests each of the component sets of this union, they are already simple sets. Therefore, the list of the tabulated sizes is returned and the level one execution continues.

For each of the remaining levels, a rule exists which says, for example, that the corresponding dosupx goal succeeds if the desired level is not equal to x. In this case only the simple test for level five fails and backtracking must occur to satisfy the 'dosup5' goal. The second 'dosup5' rule contains the calculation needed to obtain the appropriate upper bound. The two 'dosup5' rules are shown in figure 2.1. In these rules L is the intersection set list and I is the level of knowledge requested by the user. The goal established by the 'dosup' rule is the following:

```
dosup(size,and([male,or([chol(1),chol(2)])),_,Val,5).
```

The PROLOG system will match this goal to the first rule in figure 2.1 and instantiate the variable L to the set list and the variable I to 5. The first subgoal which this rule will then attempt to prove will be not(5=5) which of course will fail, causing the system to backtrack to the next 'dosup5' rule. This rule instantiates the same two variables but the first subgoal, 5=5, succeeds this time and the second subgoal is then established. The 'data(attributes,Alist)' subgoal retrieves a list of the attribute names for the given database instantiated to the variable Alist. This list is provided in the 'attrlist'

file which contains all the database specific information needed by the system and which was consulted at the beginning of execution. The next subgoal set up by the system is the 'map' subgoal which causes the 'sumfreqs' rule to be executed with each element of the Alist as its first argument. The rules for the 'map' functional and for the 'sumfreqs' clause and its subordinate clauses are shown in figure 2.1.

The 'sumfreq' goal is first set up with patno as the value of the Attr variable and [male,or([chol(1),chol(2)])] as the value of the variable L. The 'sumfreqs' goal first establishes the subgoal 'getfreqs5' which returns a list of the two level five frequency distribution lists for the patno attribute for the two sets in the list. Note that the success of this clause is only possible because the lower level call to the 'dosup' rule made in the level one calculation with or([chol(1),chol(2)]) as the argument set converted this set to a simple set with respect to level five. In other words this lower level call asserted an upper bound on a level five tagged frequency distribution list for this union for all the database attributes.

Next, the system passes this list of frequency distributions on to the 'findsum5' rule. This rule is the heart of the level five calculation. It is a recursive rule whose basis is the condition that the first frequency distribution in the list of frequency distributions has been reduced to the empty list. On each iteration, the 'findsum5' rule considers the first item in the first frequency distribution list and calls the 'minfreqofitem' rule to return the minimum frequency with which this item occurs in any of the frequency distributions of the list. This minimum frequency is then added onto a running sum and it is also inserted into a new list which will eventually represent the upper bound on the tagged frequency distribution of the input

intersection set. Then the 'findsum5' rule is called recursively with the remainder of the first distribution substituted for the original distribution. When the basis condition for this recursion is satisfied, the running sum will represent an upper bound on the size of the intersection because only items which occur in the first set of the intersection can occur in the intersection set. Those that do occur in the first set can occur in the intersection set with a frequency no greater than the frequency with which they occur in that component set in which they occur the least frequently. Therefore the sum of these minimum frequencies represents an upper bound on the size of the intersection set. Note also that the new list representing the upper bound on the level five distribution of the intersection is asserted into the system cache to convert this intersection into a simple set with respect to level five. In the case of this query this conversion is not really necessary because the intersection set is the higherst level set for this query. However, the conversion is completed to maitain the generality of the 'dosup' rules. (An alternate approach to this problem is presented in chapter IV.)

The concept of a union or intersection being converted into a simple set is fundamental to the execution of this system. When the upper level 'supall' rule returns from its execution, not only will it return a list of upper bounds on the sizes of the component sets in the input set list, but it must have caused the additional side effect of tabulating the appropriate level information for each of the component sets into the system cache. In this way, when the 'dosup5' rule is called with the input set list, the system is assured that level five information is in fact tabulated for every set in the input list.

The above discussion applies regardless of the level of knowledge specified by the user. The level x dosupx rule

```
dosup5(size,and(L),_,Val,I)  :- not(I=5).
dosup5(size,and(L),_,Val,I)  :- I=5,
          data(attributes,Alist),
          map(sumfreqs,Alist, L ,Vlist),
          min1(Vlist,Val).

map(_,[ ],_,[ ]).
map(Functor,[Arg1|Arglist],Otherargs,[Val|Others]) :-
          makelist(Functor,Arg1,Otherargs,Val,Function),
          X =.. Function,call(X),
          map(Functor,Arglist,Otherargs,Others).
makelist(F,A1,Otherargs,Val,Resultlist) :-
          append([F|[A1|Otherargs]],[Val],Resultlist).

sumfreqs(Attr,L,Val) :- getfreqs5(L,sup,Fl,Attr),
          findsum5(Fl,Val,Al),
          asserta(cache(freqdist5,and(L),Attr,Al,sup)).

getfreqs5([ ],_,[ ],_).
getfreqs5([First|Rest],E,[F|Fl],Attr) :-
          isdata(freqdist,First,E,F,Attr),
          getfreqs5(Rest,E,Fl,Attr).

findsum5([[ ]|Others],0,[ ]).
findsum5([[Im,Freq]|Rest]|Others],S,[[Im,Min]|Al]) :-
          minfreqofitem(Im,Freq,Min,Others),
          findsum5([Rest|Others],Old,Al),
          S is Old + Min.

minfreqofitem(Item,Sofar,Sofar,[ ]).
minfreqofitem(Item,0,0,_).
minfreqofitem(Item,Sofar,Min,[First|Rest]) :-
          qmember(Item,First,New),
          min(Sofar,New,Sofar2),
          minfreqofitem(Item,Sofar2,Min,Rest).
```

Figure 2.1 PROLOG Code for the Level Five Upper Bound Calculation for Intersection Sets.

will always return an upper bound for the input set and will always cause the side effect of asserting into the system cache the level one and the level x information about the input set thus converting that set into a simple set with respect to level x. An analogous construction exists for the doinf rules. The general 'doinf' rule calls 'doinf1' through 'doinf5' with only the level one and the specified level calculations being executed. The level one 'doinf' rule calls the 'infall' rule which returns a lower bound on the size of all the component sets and causes the side

effect of tabulating lower bounds on the level 1 and thespecified level information about the input set.

We have thus far avoided discussing the calculation of the estimate for the size of the input set. Recall that the minimal cover set used to calculate the upper bound on the example set was different from the maximal union covered by the queried range used to calculate the lower bound. In order to obtain a truly meaningful estimate as to the size of this set, we must calculate an estimate for both of the above sets and average the results. For this reason, the 'doest' rule is called twice, once with each of the two forms of the input set used as its argument. The execution of the 'doest' rule is quite similar to the 'dosup' and 'doinf' rules except that only level one estimates are calculated. The 'doest' rule calls the 'estall' rule for the input set list which returns an estimate as to the size of each of the sets in the list. The 'estall' rule either returns the tabulated size of each component set or calls the 'doest' rule recursively for sets which have no tabulated size estimates available. Of course, when each of these lower level 'doest' calls returns, the argument set has an estimate as to its size recorded in the system cache. Because the estimate rules were the same for all levels of knowledge, these calculations did not produce very interesting results. For that reason, the estimates are not discussed at all in chapter III.

So far we have only discussed with any measure of detail the implementation of the rules for level five upper bound calculations. The general structure of the calculation for the remaining bounds calculations is similar, however, there are some significant differences. First of all, there are no rules to calculate new upper bounds for levels two through four union sets and new lower bounds for levels two through four intersection sets. Therefore, the 'do' rules

for these levels merely use the already calculated level one bounds and cause the side effect of converting the input set to a simple set with respect to the given level. Regardless of whether they calculate a new bound on the size of the input set all of the level calculations above level one involve a call to the 'map' functional to execute a given calculation for each attribute in the attibutelist. The calculations are also quite similar. Each one involves a retrieval rule which makes a list or lists of the information in the database abstract corresponding to the input sets. Then a recursive function passes through this information list building a bound value for the size of the set and a bound for the appropriate level information corresponding to the set. (If the calculation corresponds to one of the levels where no new bounds are possible, this recursive function calculates only a bound on the appropriate level information.) This level information is then asserted into the system cache and the calculation is repeated with the next attribute. When the map functioanl returns its list of answers, the sup rules choose the minimum from the list as the least upper bound while the inf rules choose the maximum from the list as the greatest lower bound. For a thorough discussion of the equations used in these rules, refer to [Ref. 2].

E. SUMMARY

This chapter has discussed in detail the implementation of the Database Abstract Query Estimation System, including its two subsystems, the database abstract generation subsystem and the query estimation subsystem. The following chapter will present the results of various test cases executed on this prototype system and discuss the concept of developing the optimum database abstract.

III. ANALYSIS OF SYSTEM TESTS

A. TESTING STRATEGY

System testing of the DAQUES system was not conducted in a completely rigorous manner, however, sufficient test cases were chosen to demonstrate the strengths and weaknesses of the system and to show how the system can be customized to suit the requirements of different users. Three different factors were varied during the tests to demonstrate several characteristics of the system. These factors were the level of knowledge of the database abstract, the granularity of the database abstract, and the use of partition aliases to construct the database abstract. The ten queries listed in table I were run on each of the test database abstracts to determine the effects of varying the above factors. Inspection of table I will show that the queries used represent a good variety of different sets that the user may query. Of the ten sets in the table, five are intersections and five are unions. Some of the sets include a large percentage of the total database while others represent only very small sets. A union of three sets was chosen as one of the ten sets to demonstrate the system's ability to handle larger unions or intersections. Each of the database attributes was used several times in the ten queries to provide a more even mix of test cases.

Three characteristics of the system execution were compared during the tests in order to arrive at useful conclusions. The first of these characteristics was tightness of the upper and lower bounds calculated by the system. This measurement apperared to be the most intuitive and easily observed indication of how accurate the system

TABLE I

Ten Queries Used to Test the System

Query Number	Set	Actual Size
1	da(12,26) + temp(36.9,37.3)	290
2	chol(115,277) * pred(3,20)	340
3	patno(17,47) + da(17,47) + pred(17,47)	385
4	temp(37.8,40.8) * male	8
5	female * chol(129,245)	244
6	patno(3,44) + da(30,95)	327
7	patno(25,27) * da(13,18)	17
8	chol(115,230) + pred(22,30)	435
9	temp(36.8,37) * patno(44,47)	13
10	pred(12,17) + da(19,20)	101

NOTE(1) The first query can be paraphrased as the set of all patients who either have a disease activity between 12 and 26 inclusive or have a temperature between 36.9 and 37.3 inclusive. The second query refers to the set ofall patients who have both a cholesterol level between115 and 277 inclusive and prednisone level between3 and 20 inclusive.

NOTE(2) Query number 4 is also referred to as:
hitemp * male
and Query number 6 is referred to as:
lowpat + hida
when using the aliased database abstract.

responses were. The second characteristic measured was the response time of the system for the given input query. This measurement used the PROLOG 'cputime' function which in turn uses the Unix operating system time utility. Therefore the response time measurement may vary plus or minus several seconds depending on the user load on the system. The test cases were all run under lightly loaded conditions to minimize the adverse effects of these discrepancies. Finally, the third characteristic measured as output was the storage requirements to hold the database abstract file. The PROLOG 'consult' function returns the size of the consulted file following a successful consult operation. This logical file size was used because it represents the the storage required by the database abstract in the internal PROLOG database.

Of course, a better system is determined by a tighter bound on accuracy, a shorter speed of execution, and a smaller storage requirement for the database abstract. The following three sections of this chapter will discuss each

of the variables in turn and describe how varying them affected the system characteristics mentioned above.

B. LEVEL OF KNOWLEDGE OF THE DATABASE ABSTRACT

The ten queries of table I were used to test this factor by running each query five times, once for each level of knowledge. Figure 3.1 shows the graph of average system accuracy and speed of execution over the ten queries tested. This graph was prepared using a database abstract with all of the attributes partitioned into four divisions, however, the general shape of the accuracy curve is identical for all other granularities tested. This curve does show only the average accuracy and response time results of ten queries, however the reader can gain an appreciation for the general tendencies of the system accuracy and response time as the level of knowledge is varied even if no appreciation is gained for the sizable variance in responses among the queries tested.

This graph shows the manner in which the accuracy of the system response as well as the speed of system execution increases as the higher levels of knowledge are used for the queries. (Note that an increase in accuracy of query response translates into a downward slope on the graph.) System accuracy increases at a relatively constant and rather slow rate for the first four levels, however the increase in accuracy takes a relatively sharp curve upward between levels four and five. One reason for this greater increase in accuracy is the fact that the queries tested included five unions and five intersection sets. Recall that the system did not have any rules for calculating new upper bounds for set unions for levels two, three, and four nor did the system have rules for calculating new lower bounds for set intersections for the same three levels.

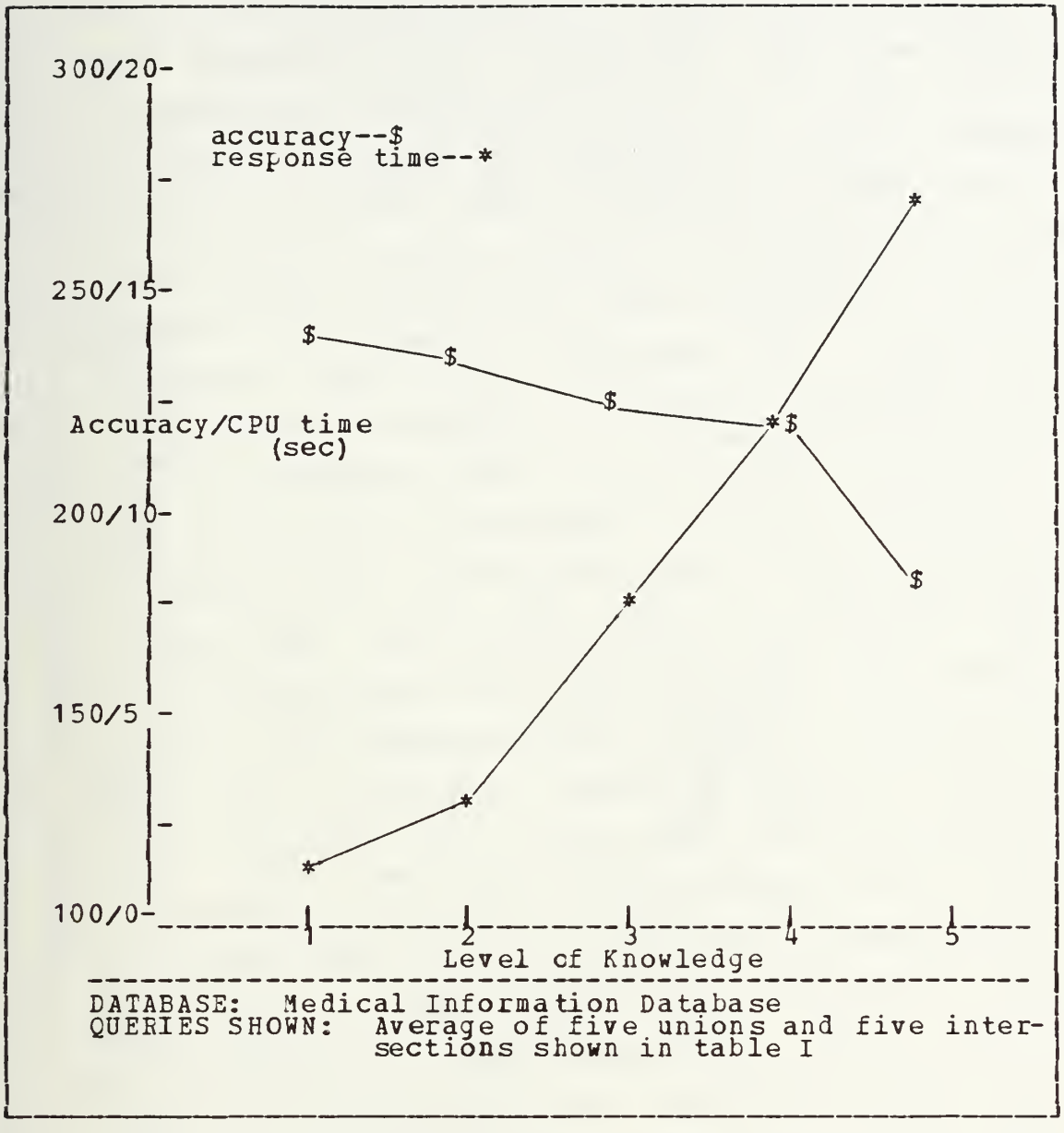

Figure 3.1 Accuracy and Speed of Response for Five Levels of Knowledge.

These levels for which news bounds could not be calculated merely used the level one bounds for that query. Therefore, It was not until level five was chosen that any given query could show an improvement over level one in both the upper and lower bounds simultaneously.

Also contributing to the relatively sharp increase in system accuracy when level five calculations were chosen is the fact that level five knowledge seems to represent a sharp increase in the amount of the original information content of the database retained by the database abstract. The lower level database abstracts contain information about the frequency of occurrence of certain values in each partition set, but no information about the values to which these frequencies refer. The usefulness of comparisons between the database abstract information stored about component sets in a union or intersection is limited because there is no way of knowing to which values these frequencies refer. In some cases, worst case bounds must assume that the frequencies refer to the same value and in other cases the worst case assumption is that the sets are disjoint. Because bounds must be calculated using worst case assumptions, the lack of information about the values to which the database abstract frequencies refer results in a serious degradation in accuracy of system responses. The tagged frequency lists of level five contain these values and associate them to the frequency with which they occur. This added tag information appears to raise the real information content of the database abstract to a plane much higher than the previous levels. These results raise the interesting question of whether an intermediate level of knowledge ought to contain the mode value as well as the mode frequency and the second most frequent value together with its frequency. Perhaps, rules more similar to the level five rules could be developed to improve the accuracy of the system responses without a significant degradation of system response time. This question is discussed in more detail in chapter IV.

The second curve of figure 3.1 shows the steady increase in the time required to complete a system query as the level of knowledge is increased from one to five. The increase in

this system characteristic is much more constant than the increase in system accuracy. This trait indicates that the jump in accuracy encountered in using level five information is not matched by a similar increase in response time. However, figure 3.2 shows that the storage area required for

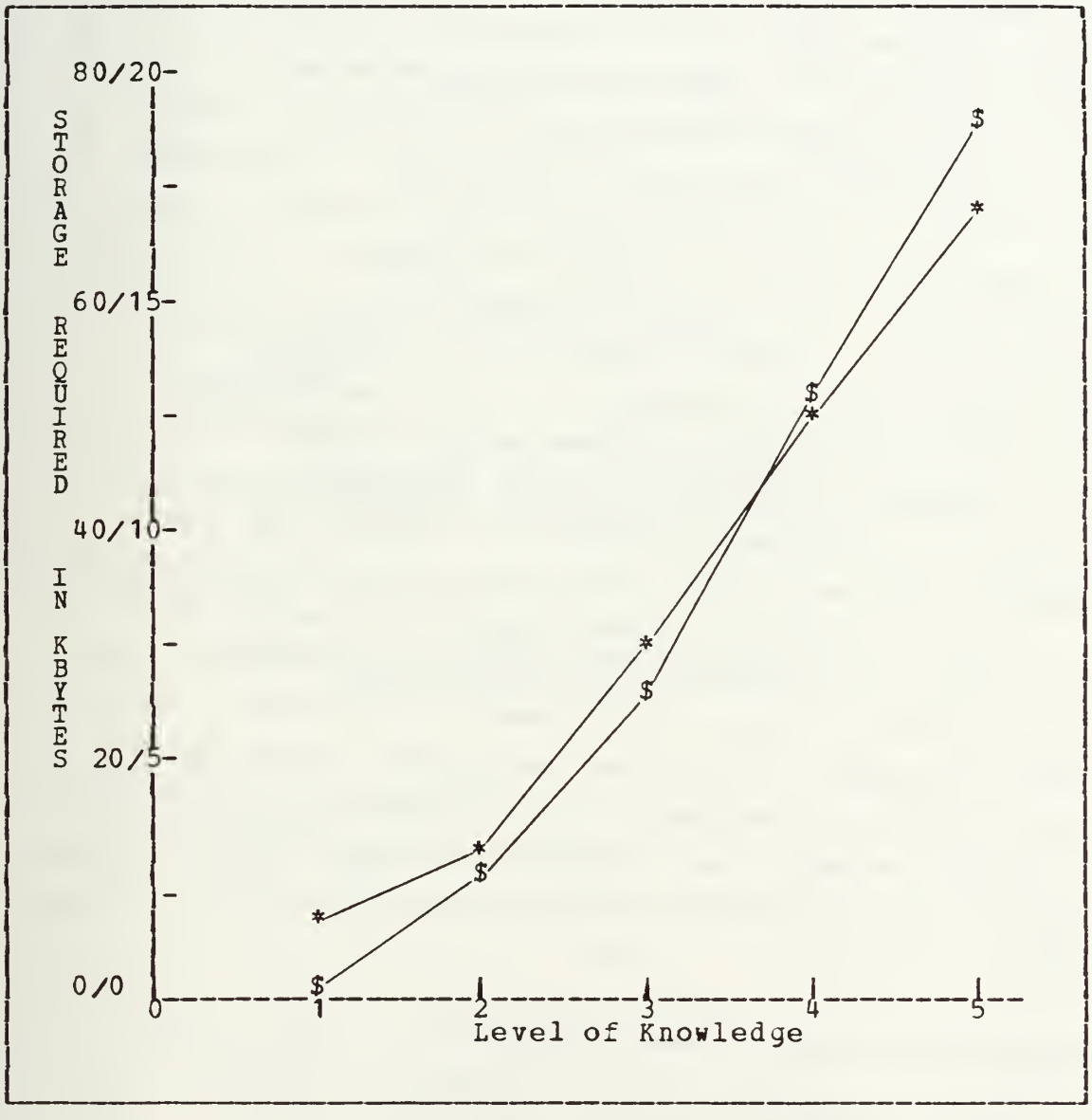

Figure 3.2 Storage and Response Times for the Various Level DBAs of the Medical Database.

the database abstracts for the various levels of knowledge also increases at a constant rate for levels one through five. The average response time for the ten queries is also shown on figure 3.2 to demonstrate that both of these characteristics increase at relatively the same rate as the level of knowledge is increased. Figure 3.3 shows a plot of storage required for the various database abstracts versus the average response times of the ten queries tested for the same database abstracts. This figure demonstrates that the increase in both storage requirements and response times is relatively constant as the level of knowledge is increased from one to five. We can conclude from this graph that the tradeoff made for greater accuracy at higher levels of knowledge results in comparable penalties of increased response times and database abstract storage requirements.

From the preceding discussion, we can conclude that if the user needs relatively tight bounds on the size of the set being queried and is not overly concerned with the response time or the memory required to execute the query, level five is most definitely the optimum level of knowledge to use. However, if he is limited severely by the amount of memory available in which to store the database abstract, he may wish to choose to use a lower level of knowledge. With this option, however, he will suffer a large penalty in the accuracy of the answer calculated by the system. If memory is not a major concern, the user will obtain the most accurate results per second of response time using level five knowledge.

C. GRANULARITY OF THE DATABASE ABSTRACT

1. Varying Granularity of All Attributes

The decision as to which level of knowledge to use for a given query is one which is made by the user. Because

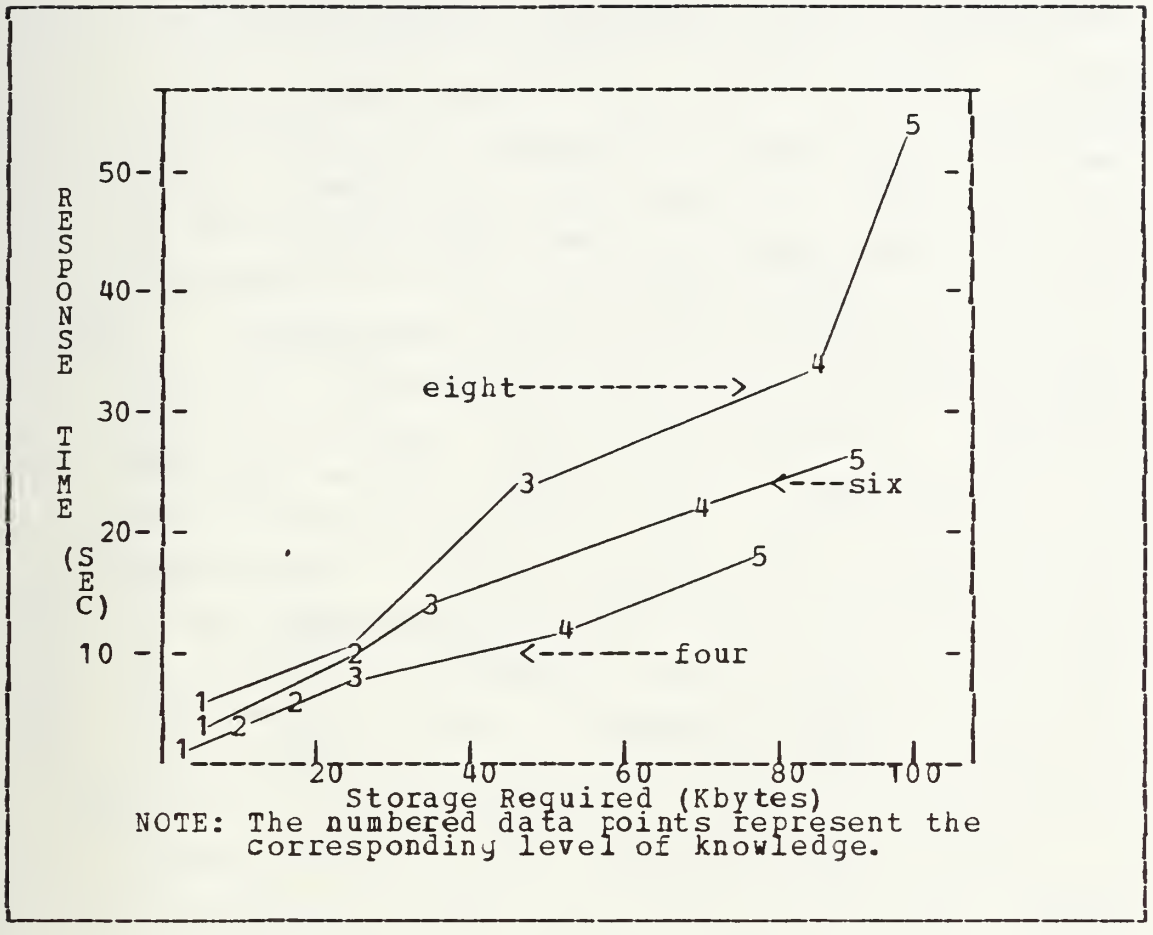

Figure 3.3 Space Versus Time Plot for Various Granularities.

all five levels are available tc the user at any given time, the user may choose one level fcr one query to obtain satisfactory results and a different level for a subsequent query. On the contrary, the decision as to the granularity of the database abstract is made by the database administrator and cannot be changed unless the database administrator recalculates the entire database abstract from the original database. Because this system is primarily designed to be used with very large databases or when access to the actual database is not available, this decision is much more permanent than the one discussed in the previous section.

The database abstract generation subsystem of DAQUES is capable of constructing a database abstract with a different granularity used for each of the numeric attributes. We will attempt here to discover what characteristics of a particular attribute tend to make a finer granularity more advantageous in terms of accuracy and to make that improved accuracy worth the penalty of increased system response time and increased storage requirements.

Figure 3.4 shows the differences between the upper and lower bounds, averaged over the ten queries, for the five levels of knowledge. The different curves represent granularities of two, four, six, and eight partitions for all of the attributes in the sample database. These curves demonstrate that the accuracy of results increases significantly as the granularity of all of the attributes is changed from two to four. However, the increase in accuracy as the granularity is shifted from four to six and from six to eight is less significant. Figure 3.5 shows that the increase in average system response time is fairly constant as the number of partitions in the database abstract is increased from two to eight. Figure 3.6 shows that the increase in storage required for the various database abstracts is relatively constant as the granularity is made finer.

Figure 3.11 shows the storage requirements of the various mixes of the database abstract for levels of knowledge one through five. From this figure we can see that partitioning an attribute more finely results in a very predictable increase in the size of the database abstract. For levels one through three doubling the number of partitions for the numeric attributes nearly doubles the storage requirements of the database abstract. For levels four and five, doubling the number of partitions results in a more finely partitioned database abstract approximately one and

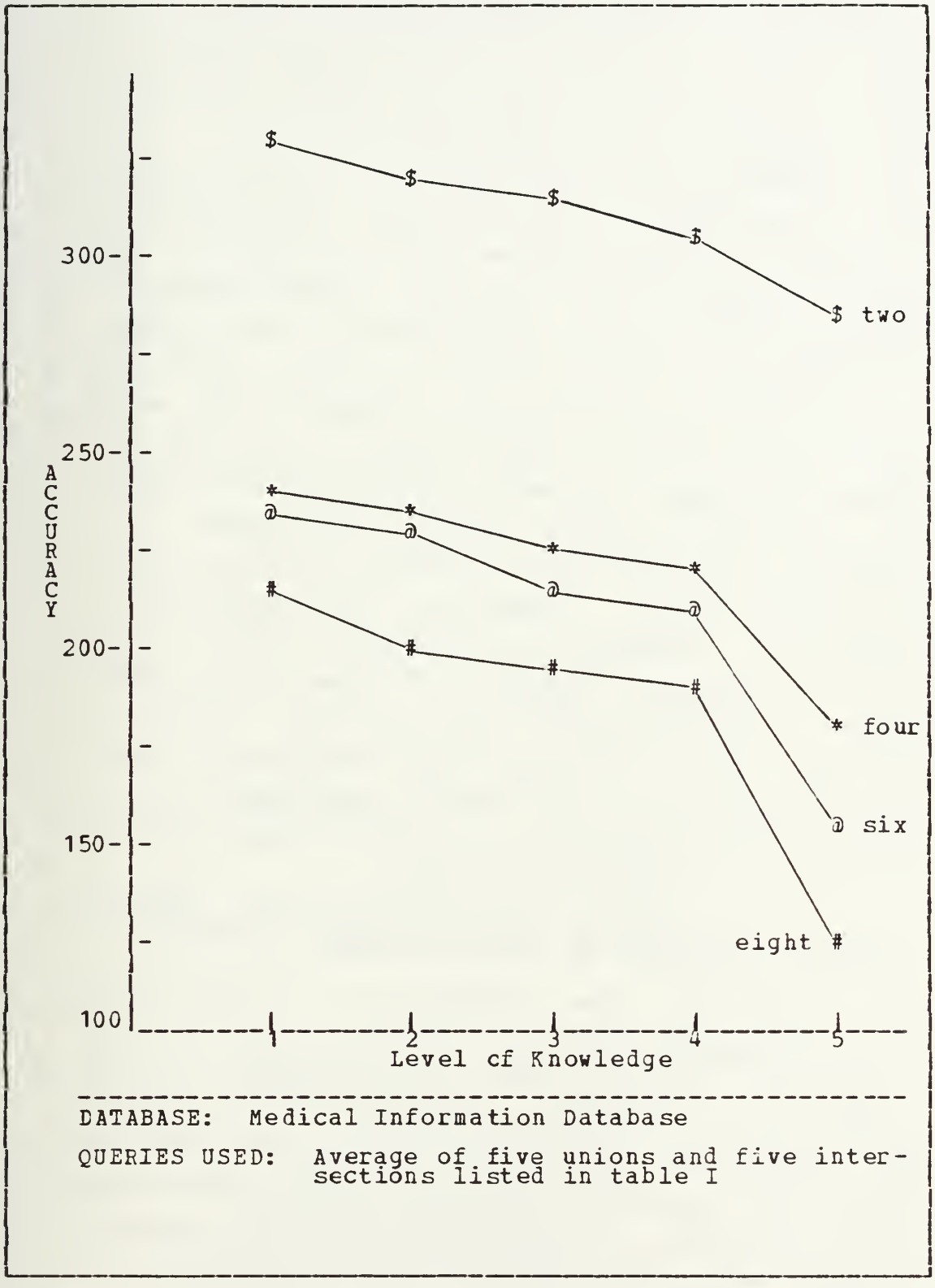

Figure 3.4 Accuracy for Various Granularities.

one half times the size of the more coarsely partitioned one. This smaller increase in the storage requirements at higher levels is due to the fact that many of the values in the frequency lists of these two levels are merely redistributed as the attibutes are partitioned more finely.

From these curves we can conclude that increasing the number of partitions on all attributes in the database abstract causes a steady increase in the penalties of greater system response time and storage requirements but results in smaller and smaller improvements in the accuracy achieved. Therefore it appears that at a certain point the penalties incurred by dividing the database into more partitions is not worth the small increase in accuracy of query responses obtained. Although these conclusions have been drawn from experimentation on only one databse, it appears resonable to assume that the results could be generalized to other databases. Considering this restiction on the usefulness of increasing the granularity of all the attributes of the database abstract, it seems advantageous to investigate whether dividing only certain attributes more finely and leaving the others less finely partitioned will result in nearly the same accuracy improvements for a smaller penalty in response time and size of the database abstract.

2. Varying Granularity of Only One or Two Attributes

To investigate the advantages of partitioning certain attributes more finely than others, the system was tested with seven different mixes of partitioning granularity. For each of the numeric attributes in the database, a database abstract was created with the given attribute divided into eight partitions and the remaining attributes divided into only four. Two additional database abstracts (mix and nmix) were also created, 'mix' dividing the temp

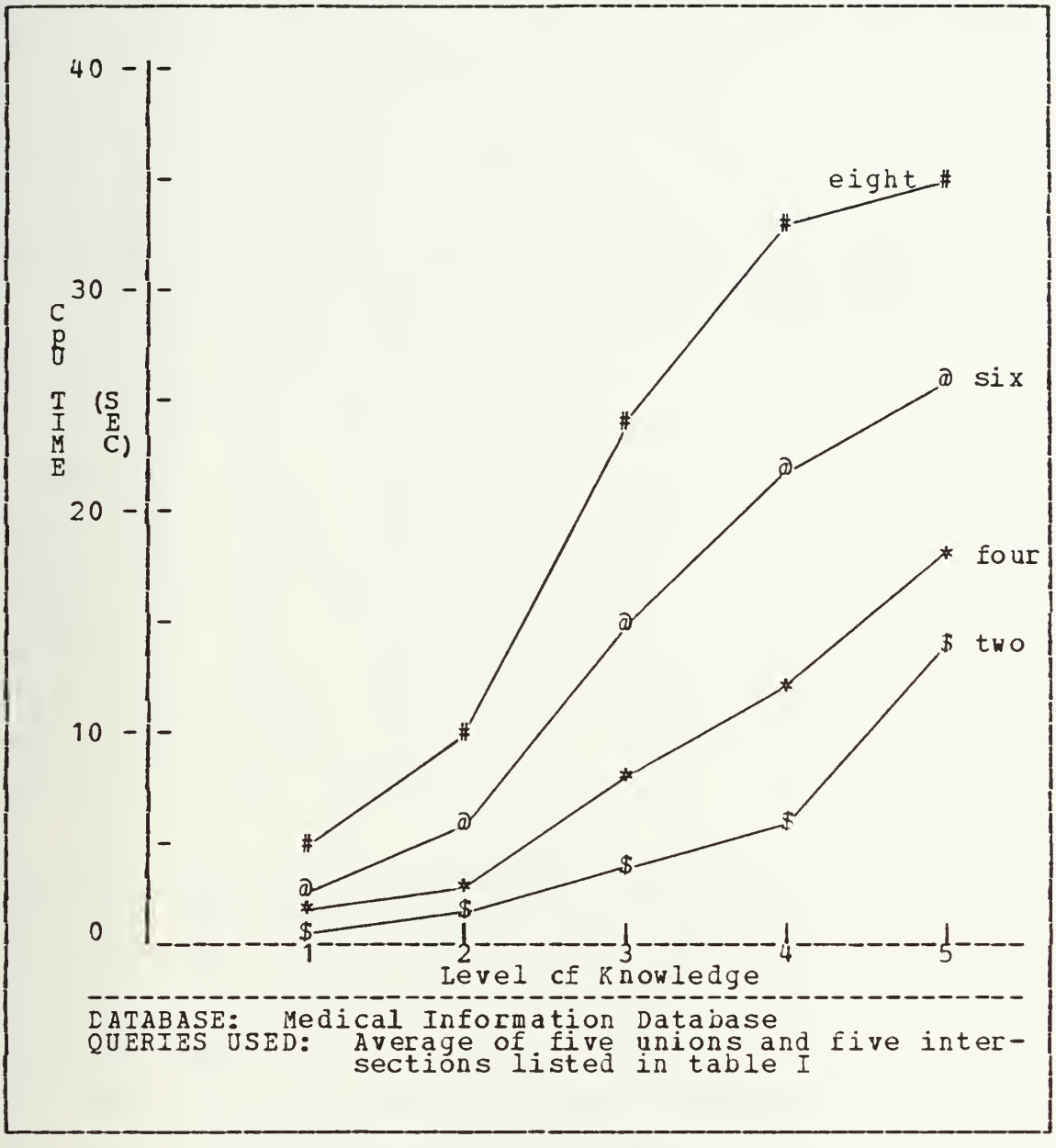

Figure 3.5 Response Times for Different Granularities.

and chol attributes into eight partitions and the remaining numeric attributes into four partitions. The 'nmix' database abstract reversed this construction resulting in four partitions for the temp and chol partitions and eight partitions for the other other numeric attributes. Table II

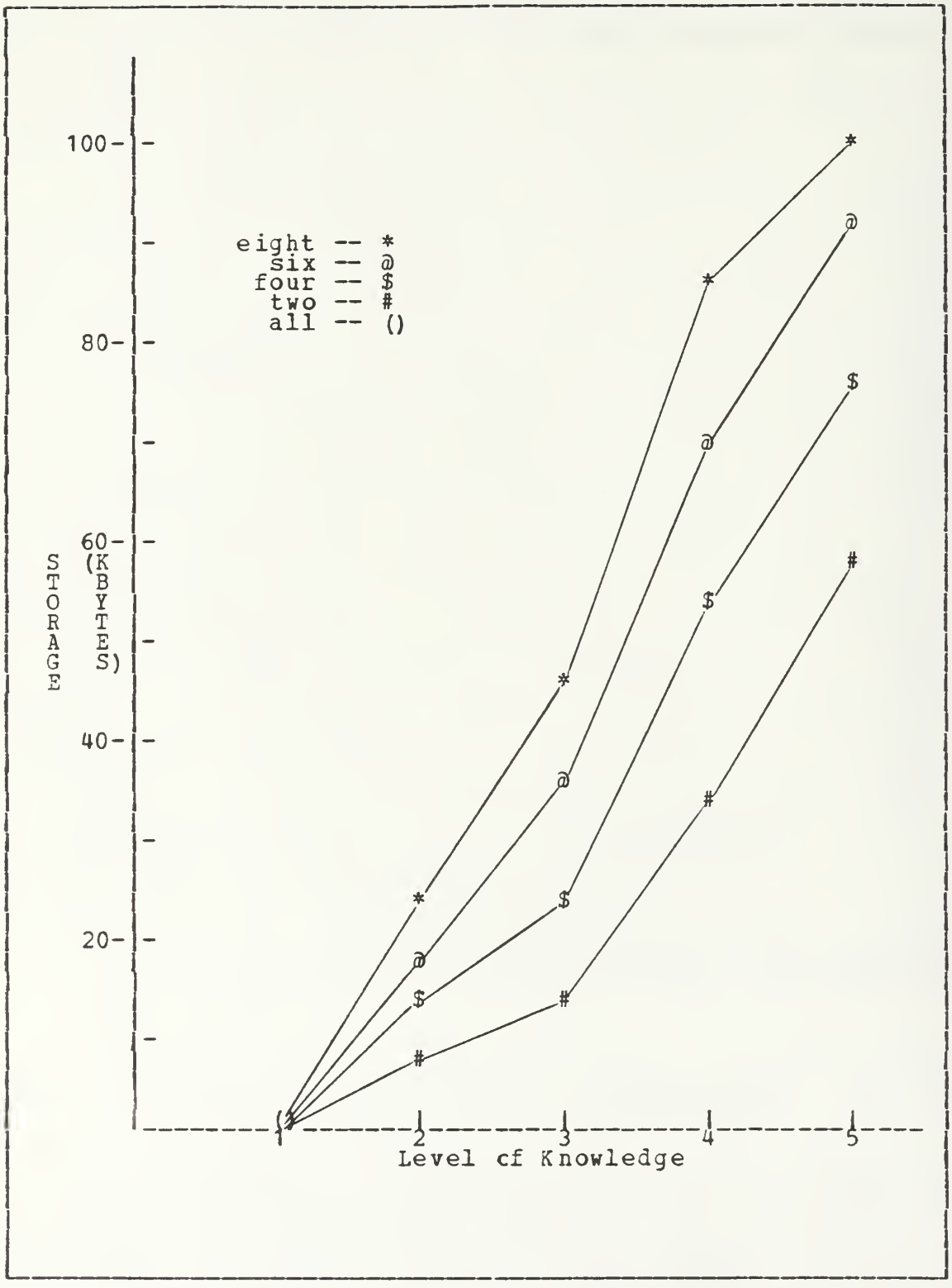

Figure 3.6 Storage Requirements of Medical Information DBAs for Various Granularities.

provides an explanation of the granularities of the various attribute partitions in each of the database abstracts named

TABLE II

Breakdown of Various Mixes of DBAs

DBA NAME	NUMBER OF PARTITIONS FOR THE GIVEN ATTRIBUTE					
	PATNO	SEX	DA	TEMP	CHOL	PRED
two	2	2	2	2	2	2
four	4	2	4	4	4	4
six	6	2	6	6	6	6
eight	8	2	8	8	8	8
mix	4	2	4	8	8	4
nmix	8	2	8	4	4	8
pamix	8	2	4	4	4	4
dmix	4	2	8	4	4	4
tmix	4	2	4	8	4	4
cmix	4	2	4	4	8	4
pmix	4	2	4	4	4	8

in the figures in this chapter. Figures 3.7 through 3.9 represent the accuracy results for three of the more interesting queries tested on this system. Each figure shows the actual upper and lower bounds calculated for all seven of these database abstracts together with the actual value of the query calculated from the original database.

Careful study of these figures shows that improvements in accuracy were shown only when one of the components of the intersection or union set being queried involved the attribute that was partitioned more finely. For example, figure 3.7 shows that the query requesting the size of the set of all males with temperatures between 37.8 and 40.8 inclusive resulted in one response from all database abstracts with the temperature attribute divided into only

four partitions and a different, more accurate response when utilizing the database abstract with the temperature attribute divided into eight partitions. This result is also present, but is less obvious in those queries where the queried set involves two numeric attributes. In figure 3.9, the query involves the patno and temp attributes and the database abstracts which partition these attributes more finely, tmix, pamix, mix and nmix all result in bounds which are tighter than the other database abstracts. However, it is interesting to note that the database abstracts which partition the patno attribute more finely result in a much greater improvement in accuracy than those which partition the temp attribute more finely. The same phenomenon is present in figure 3.8 with a finer partitioning of the pred attribute rather than the chol attribute resulting in the greater improvement.

It is suspected that because the temp and chol attributes are highly skewed (i.e. have a very large mode frequency), increasing the number of partitions for these attributes from four to eight does not have the same ability to increase the information content of the database abstract as with more evenly distributed distributions. In the highly skewed distributions, one particular value, the mode, will always occupy only one partition, possibly along with several other values. Varying the granularity of the partition cannot shift any of these mode values to separate partitions. Consequently, this very large partition, which contains a large percentage of the database tuples, will always be present and the amount of information that can be stored about it is independent of the granularity of the database abstract. Dividing the partition more finely for such attributes only results in increasing the information content of the remaining partitions which represent only a modest percentage of the entire database.

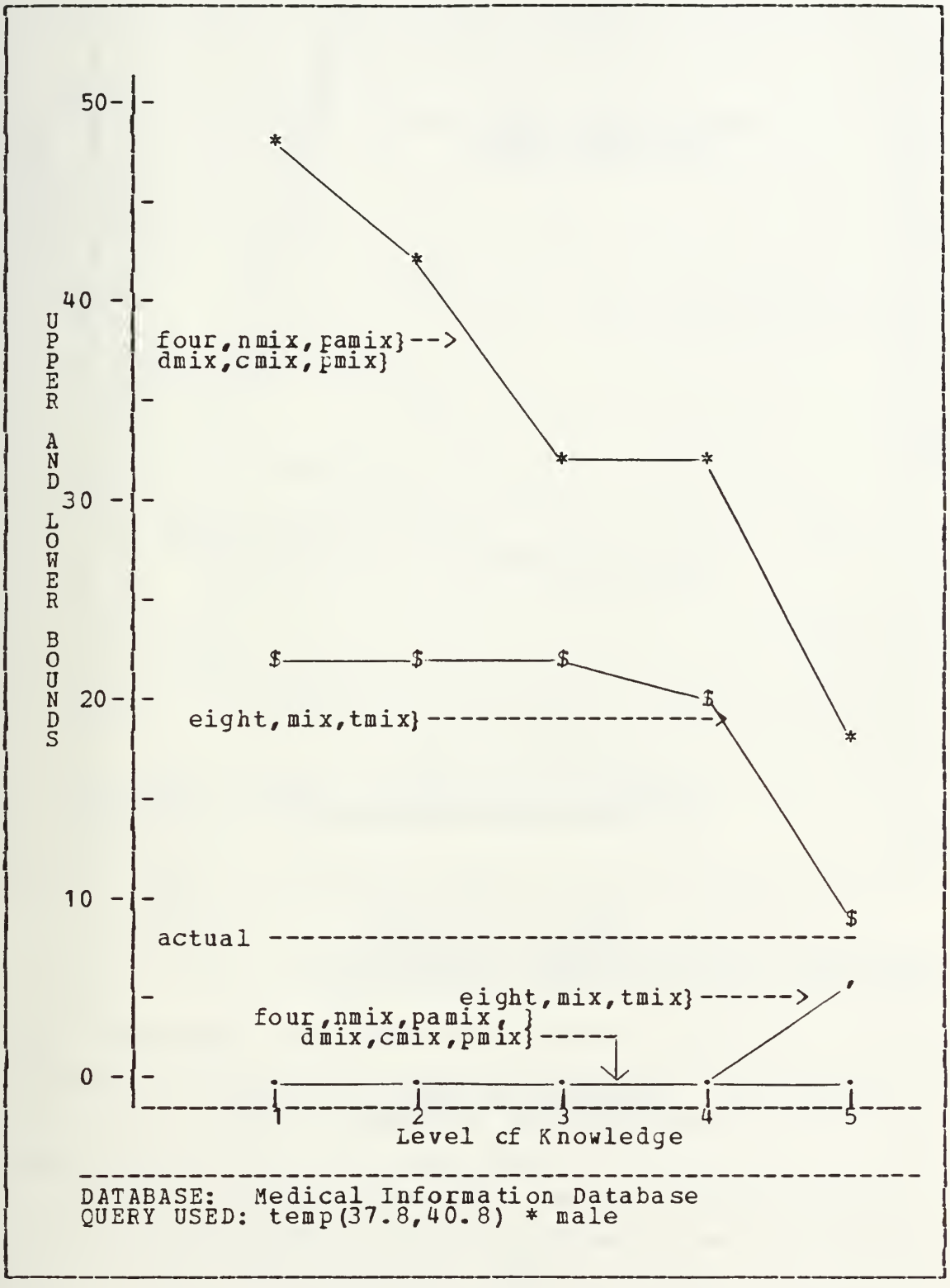

Figure 3.7 Accuracy of Query 4 Varying the Granularity of DBAs.

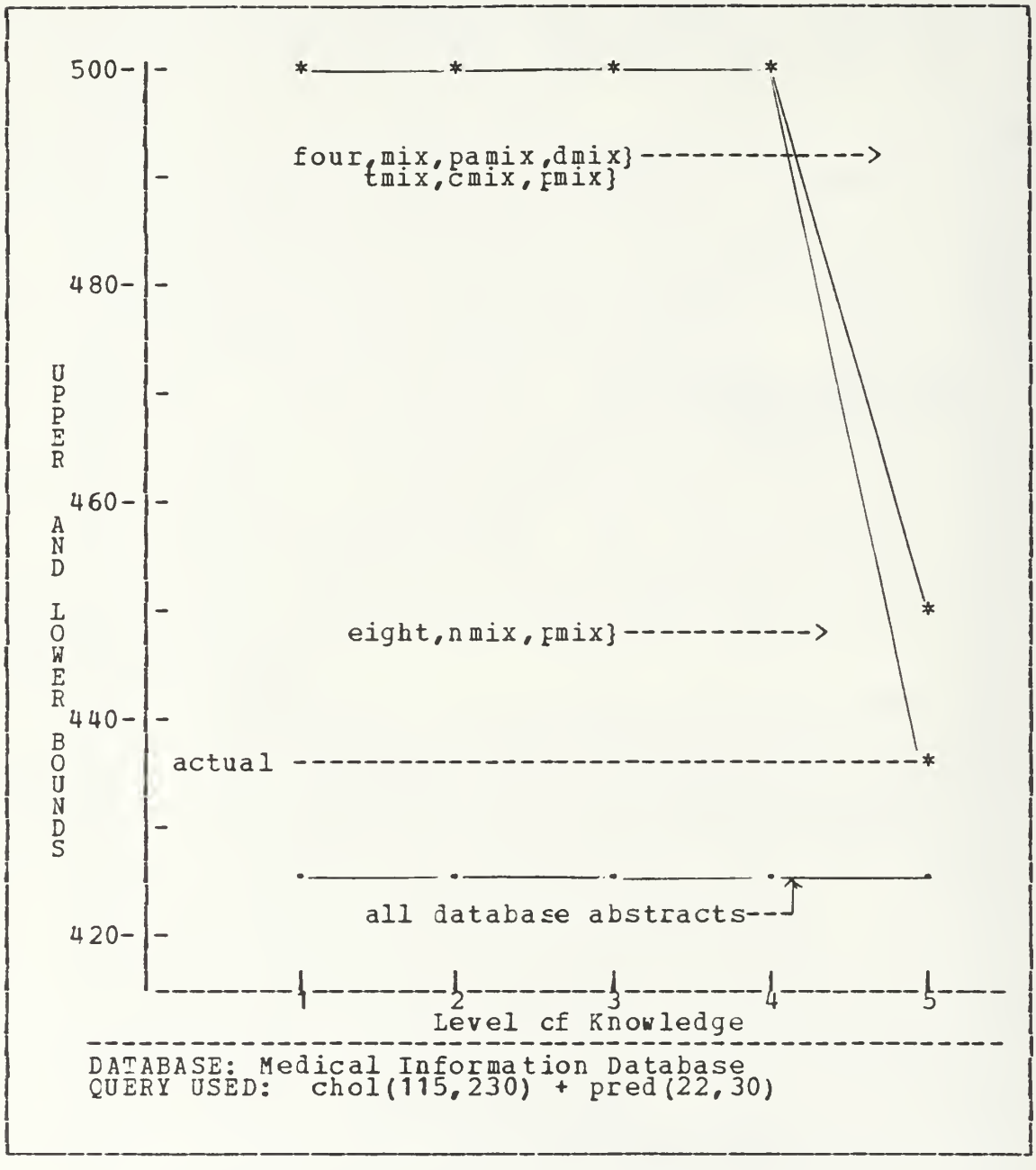

Figure 3.8 Accuracy of Query 8 Varying the Granularity of DBAs.

Another reason for the increased accuracy of system response with database abstracts which are partitioned more finely is that the union of actual partition sets chosen to

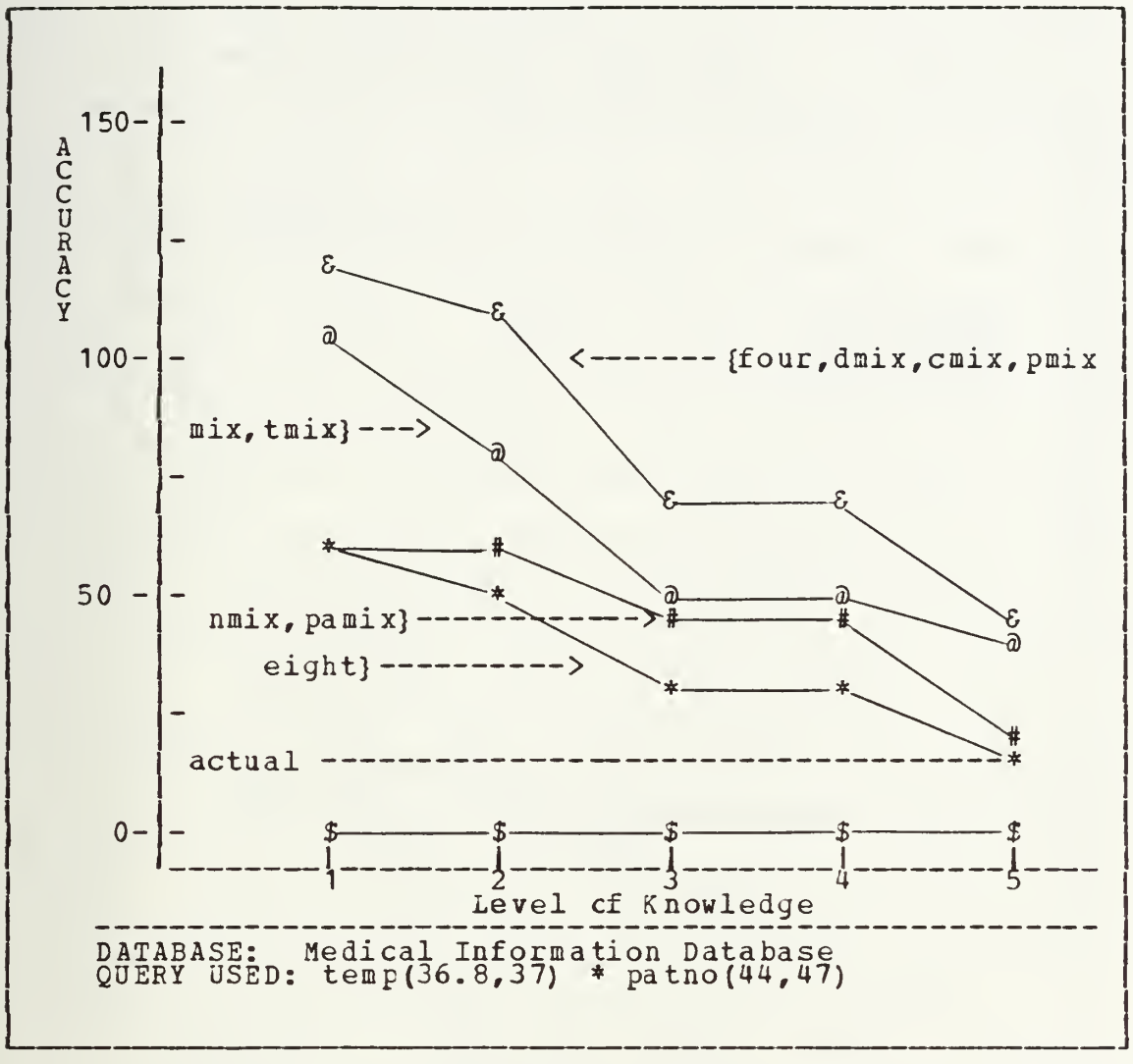

Figure 3.9 Accuracy of Query 9 Varying the Granularity of DBAs.

cover the input set defined by a range of attribute values is likely to fit that range more closely than it would using a database abstract partitioned more coarsely. Figure 3.10 demonstrates this point graphically. In this figure, we see the responses to system queries on an increasingly larger range of the numeric attribute, temperature. When the queried range is very small, say 40 to 40.8, the minimal union of actual partition sets for both database abstracts

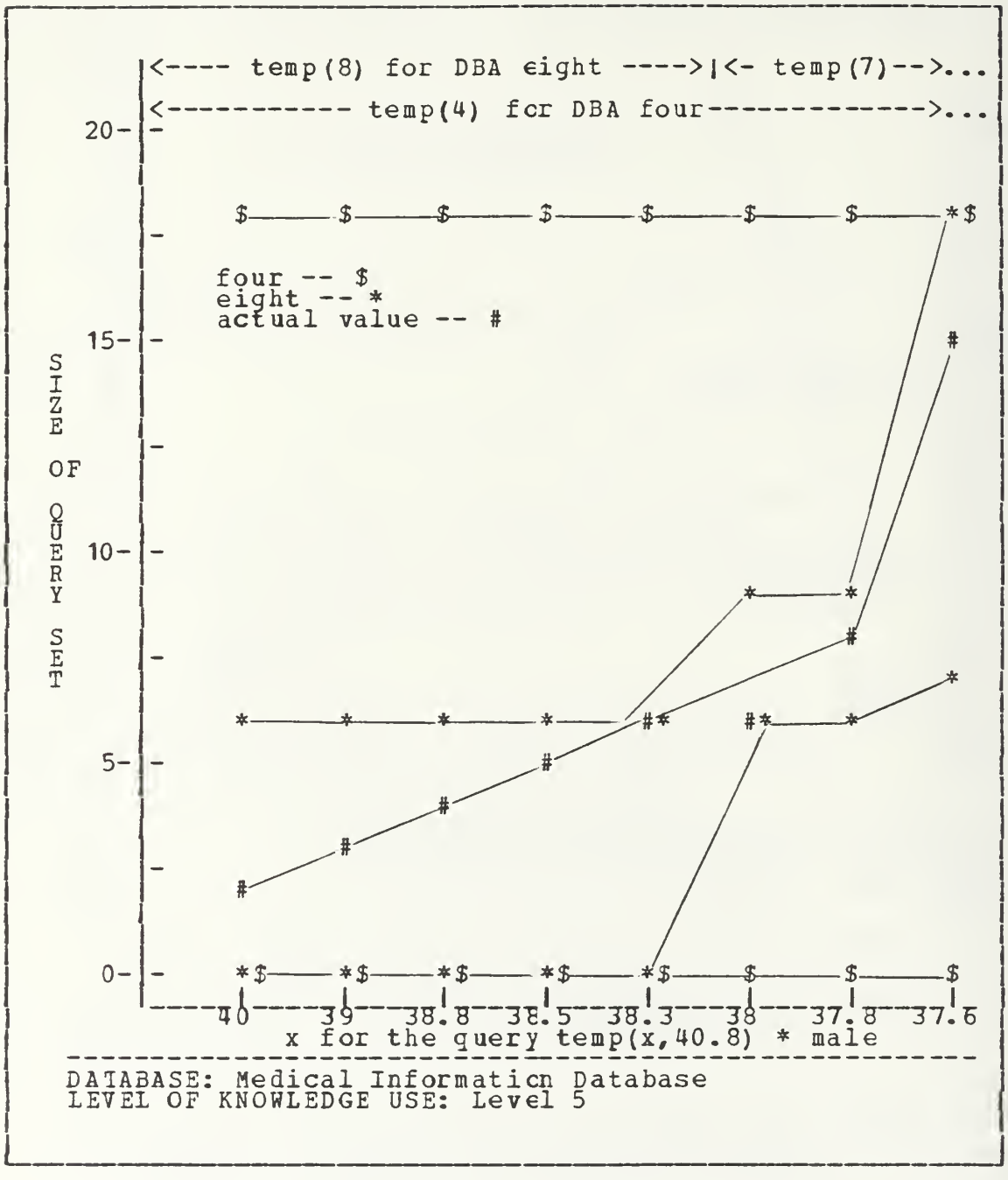

Figure 3.10 Accuracy for Two DBAs Varying the Attribute Range of the input Queries.

consists of only one set. However, the one set for the finer partition is much smaller than that for the coarser

one. Therefore the calculated upper bound for the query, which is based solely on this minimal union, is much closer to the actual size of the set when a finer partitioning is used for the database abstract. In the same way, the calculated lower bound is based on the maximal union of partition sets which is covered by the desired range. For a very small queried range, this union is the empty set, producing a lower bound of zero. However finer partitioning of the database abstract results in smaller partitons and consequently a given queried range is more likely to cover such smaller sets than the larger ones resulting from a coarser partitioning. So, for a random range queried by the user, the finer the partitioning of the database abstract, the more closely the unions actually used for calculating the upper and lower bounds will be to the queried range and consequently the more accurate the results will be. Note that figure 3.10 shows the system calculating the same upper and lower bounds for all eight query ranges tested when the coarse database abstract was used. When the finer partitioning was selected, however, the system was able to calculate several different bounds and therefore was able to discriminate between these various queries tested.

Returning to the three example queries of figures 3.7 through 3.9, if we examine the response times for these calculations, we find that the level five calculation for the query shown in figure 3.7 ranged from 6.5 to 7.5 seconds for the database abstracts which divided the temp attribute into only four partitions, while response times of 18.6 to 25.2 were experienced for those database abstracts with the temp attribute partitioned eight ways. The time 18.6 resulted from the database abstract which only divided the temp attribute into eight partitions while the 25.2 time resulted from the database abstract which partitioned all the attributes eight ways. So it seems that increasing the

granularity of even a single attribute in the database abstract will result in slower response times on the same order of magnitude as increasing the granularity of all the attributes. Note also that doubling the number of partitions in the database abstract results in considerably more than doubling the system response time. This characteristic is due to the fact that response time is not only a function of the size of the database abstract but also of the complexity of the input set. A set specified by a range such as:

male * chol(115,270)

will be preprocessed into form a) below for for a database abstract with four partitions and into form b) for a database abstract with eight partitions.

a) and([male,or([chol(1),chol(2),chol(3)])].

b) and([male,or([chol(1),chol(2),chol(3),
chol(4),chol(5),chol(6)])].

Obviously the set of form b) will require much longer to process than form a). Similarly, when only the queried attribute is partitioned more finely, the input set is still preprocessed into form b) and the complexity of the set causes the system response time to approach the response time of the system when the database abstract with all attribute partitioned finely is used. If the query set does not involve the single attribute which is partitioned more finely, then the system response time is closer to the response time of the system using the databse abstract with all of the attributes partitioned more coarsely. However, keep in mind that a finer partitioning of a single attribute will result in increased accuracy only for those queries which involve the attribute partitioned more finely. From the above discussion we can conclude that if the user

community needs additional accuracy and a single attribute predominates the system queries, it may be worth mixing the granularities of the various attributes in the database abstract to obtain the optimum accuracy at the minimum cost in system response time.

Figure 3.11 shows the storage requirements for the database abstracts of table II. Inspection of this figure reveals that increasing the granularity for only one attribute in the database abstract results in a much smaller increase in the storage required than increasing the granularity of all the attributes. Therefore, if primary memory available is a major concern, dividing of only the most commonly used attributes into finer partitions may be the optimum method of constructing the database abstract.

From the above paragraphs, we can conclude that there is no single optimum partitioning strategy which will be advantageous for all queries. However, if the database administrator is aware of a particular attribute which is involved in a large percentage of the system queries, he may decide to divide that attribute into more partitions than the other attributes. For this reason, the query subsystem of the DAQUES system allows the user access to three distinct database abstracts for his queries. The intent of this arrangement is to provide a database which is rather coarsely partitioned as the 'normal' database abstract, resulting in rather loose upper and lower bounds but more rapid response. If the user chooses the 'alternate' database abstract, he will be using one in which all the attributes are partitioned more finely but the response time for each query will be much slower. Finally, the user may choose the 'special' database abstract which is partitioned finely on one or two commonly queried attributes and more coarsely on the remaining attributes. This database abstract will result in more accurate results than the

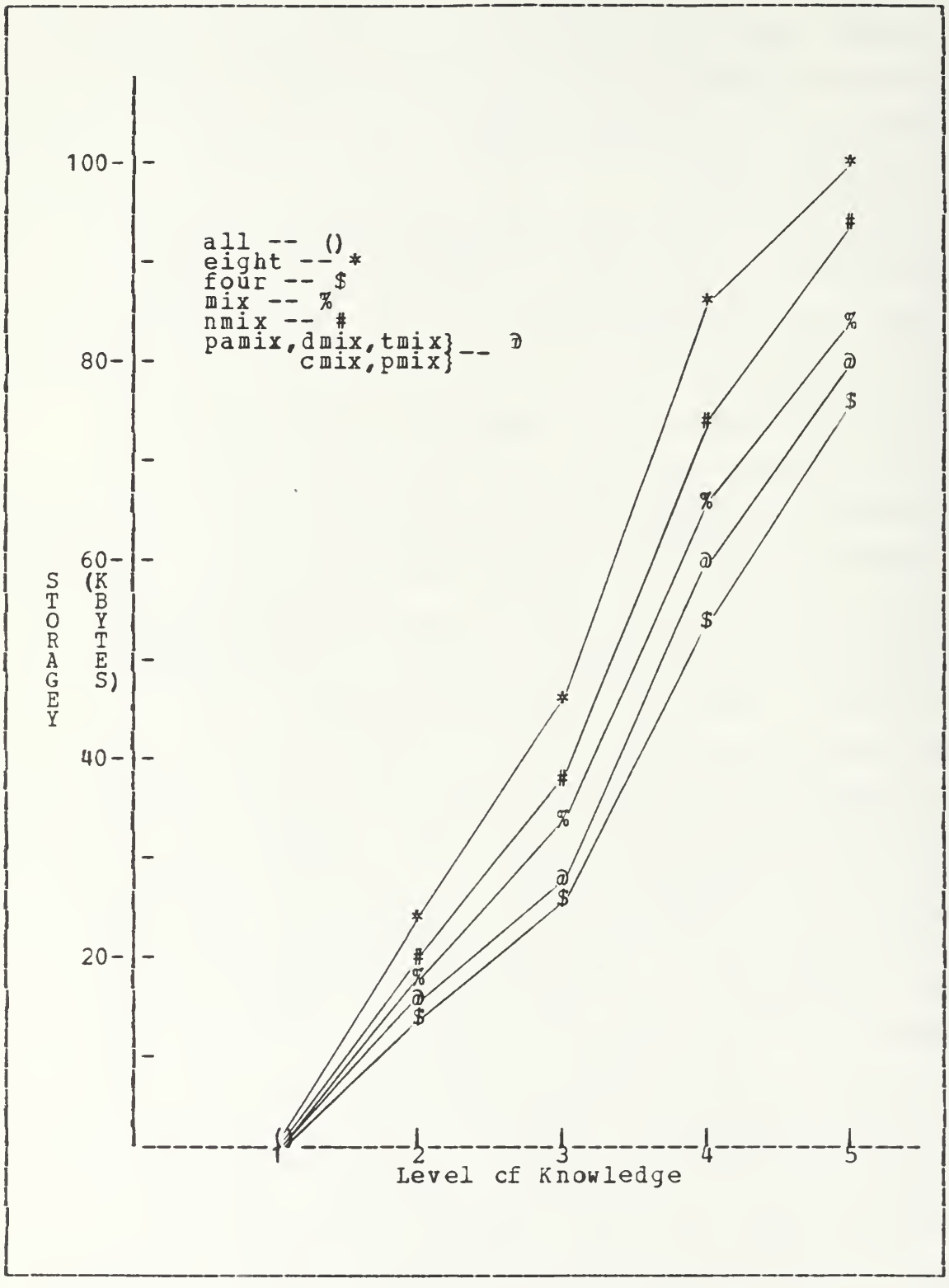

Figure 3.11 Storage Requirements for DBA's of Various Granularities for the Medical Database.

'normal' database abstract for only those queries involving the 'special' attribute or attributes and the response time will be less than that of the same query executed with the 'alternate' database abstract. Because the system only consults one of these database abstracts into the PROLOG internal database at a time, the user does not pay any added price in storage requirements other than the secondary storage required to hold the three separate database abstracts. Because secondary storage is usually readily available compared to primary storage in the form of the PROLOG internal database, this penalty seems a small price to pay for the added flexibility of this system.

D. USE OF ALIASES TO IMPROVE SYSTEM ACCURACY AND RESPONSE TIME

The previous section analyzed a method with which the database administrator could tailor the database abstract to the specific requirements of a particular group of users by partitioning more finely those attributes which are involved most frequently in the query sets of the users. This section analyzes another method of tailoring the database abstract to users by partitioning various attributes along boundaries which represent important sets in the application field and then assigning aliases to those partition sets in such a way that the aliases describe the sets. This method results in the database abstract which is the most specific to a class of users but also produces the most accurate results for queries on the special aliased sets.

The method of defining special boundaries for meaningful partition sets of a particular attribute was described in Chapter II in the section on the database abstract generation subsystem of DAQUES. The technique outlined in that chapter was used to define partitions of the patno, da and

temp attributes, assuming that the boundaries used represented meaningful descriptions of low, middle and high ranges for those attributes. The remaining two numeric attributes were partitioned using the algorithm of the generation subsystem with no attempt to produce meaningful partition sets on those attributes. Figures 3.12 and 3.13 show the query results of queries 4 and 6 of table I using the specialized database with only three partitions for each attribute. On separate curves are shown the results of the same queries using algorithmically generated database abstracts containing four and eight partitions. A comparison of these curves shows graphically the tremendous improvement in accuracy which results from using the alias technique. Because the specialized database abstract uses only three partitions for all the attributes, it requires much less storage than the database abstracts with combinations of four and eight partitions for the attributes. The system is able to evaluate queries on specific partition sets much more efficiently than it can evaluate queries on sets defined by arbitrary ranges of the attribute. This capability is caused by the method of substituting unions of actual partition sets for sets using specific ranges as described in chapter II. Therefore, the queries on the same sets executed much more rapidly with much more accurate results for this specialized database abstract than for the previous unspecialized database abstracts. These response times are demonstrated graphically in figures 3.14 and 3.15.

E. CONCLUSIONS

From the tests discussed in the previous section we can now develop several conclusions about what the optimum database abstract would be. Recall that our goals for this system, mentioned in chapter I, were to determine the

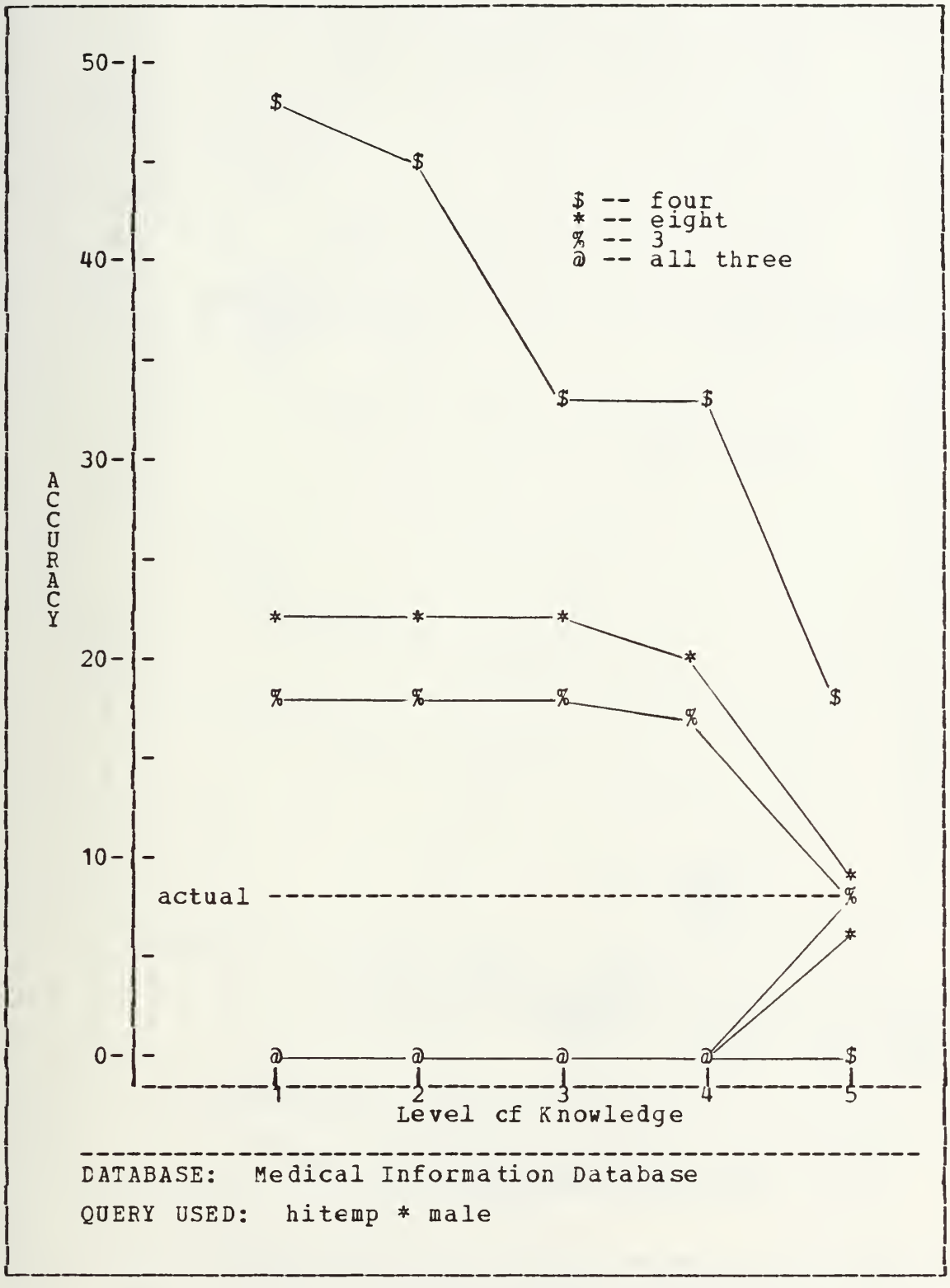

Figure 3.12 Accuracy Results for Query 4 Using the Specialized DBA.

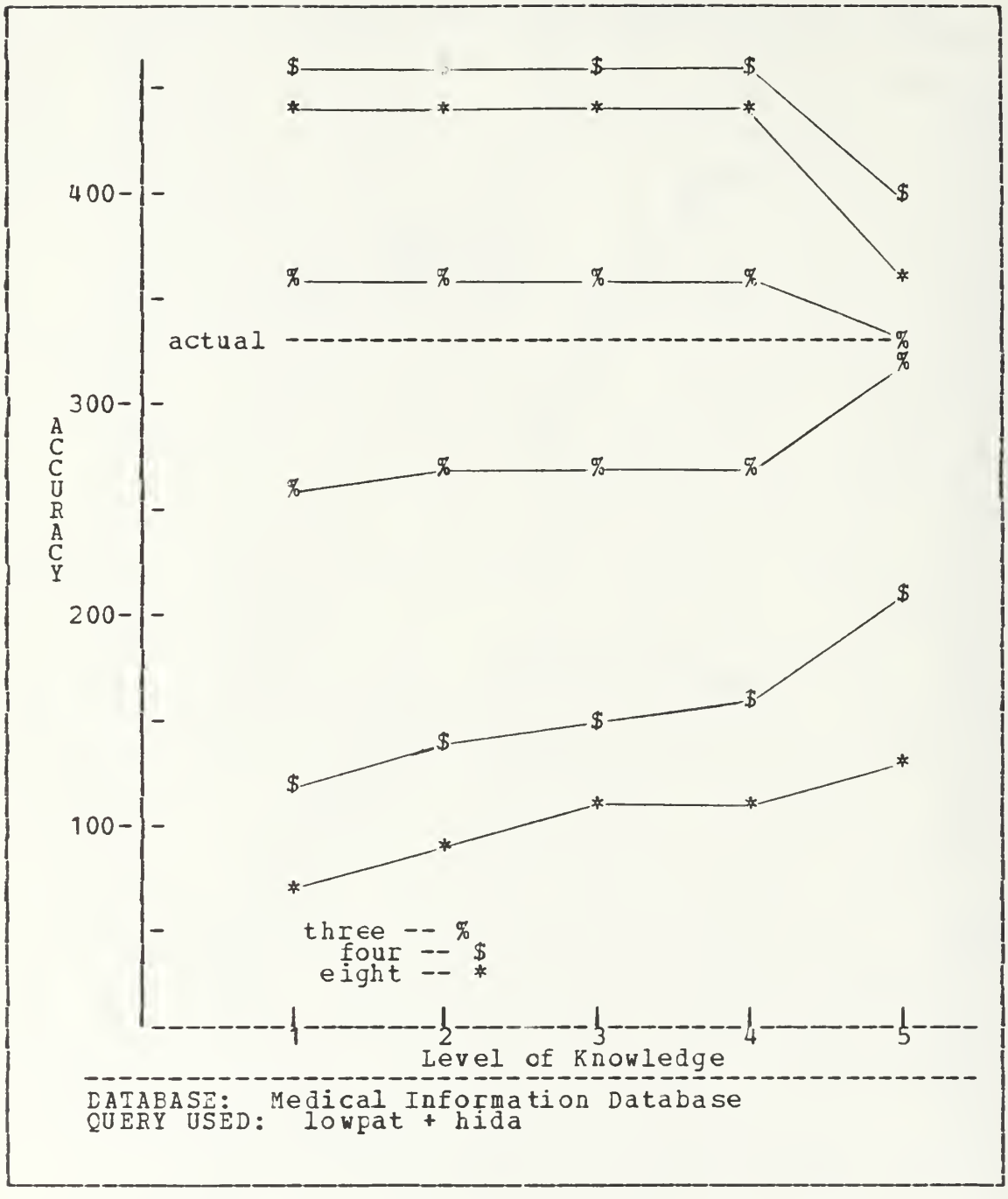

Figure 3.13 Accuracy Results for Query 6 Using the Specialized DBA.

optimum level of knowledge, and the optimum granularity of a database abstract for the query estimation system. Here,

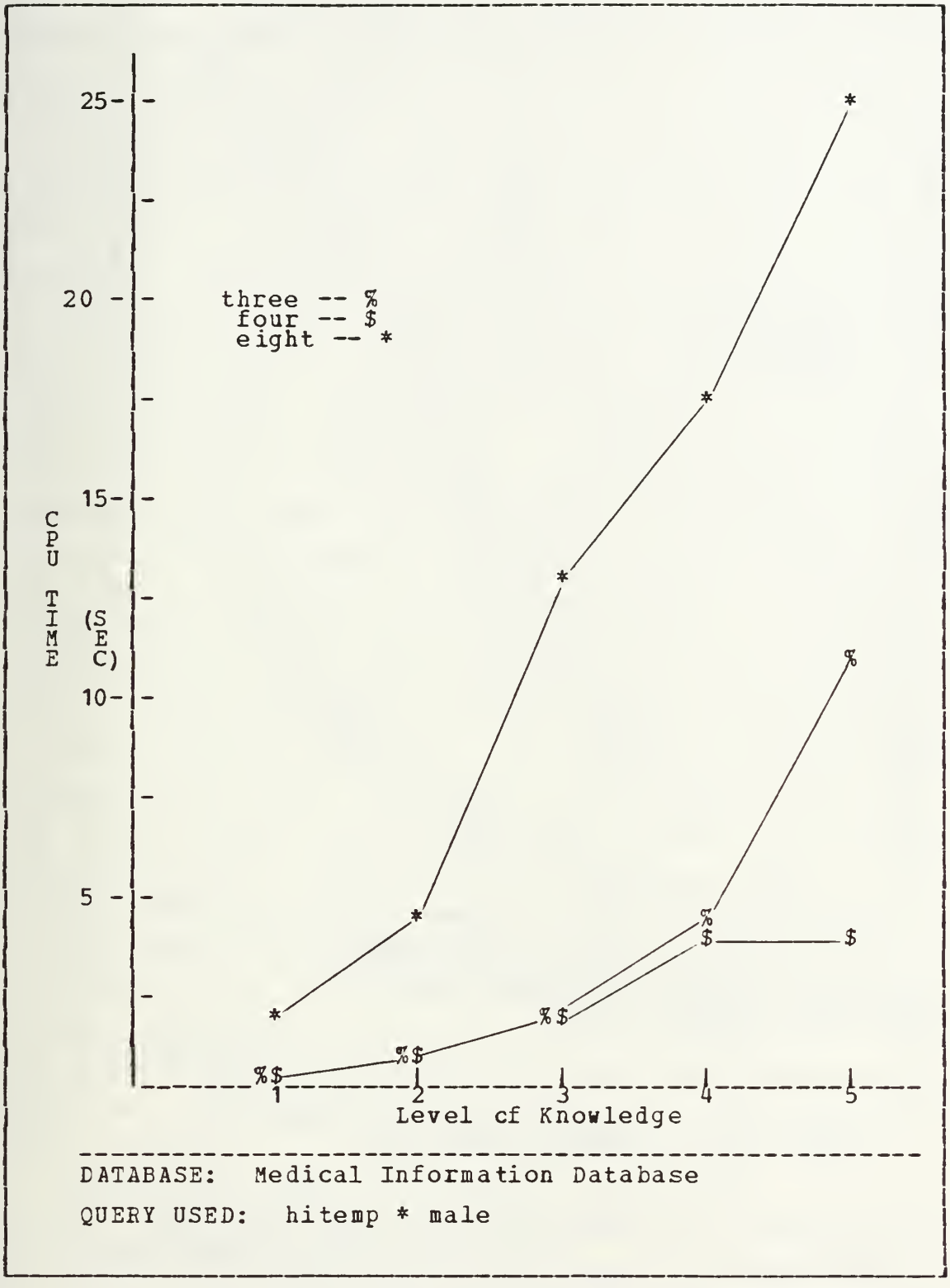

Figure 3.14 Response Times for Query 4 For the Specialized DBA.

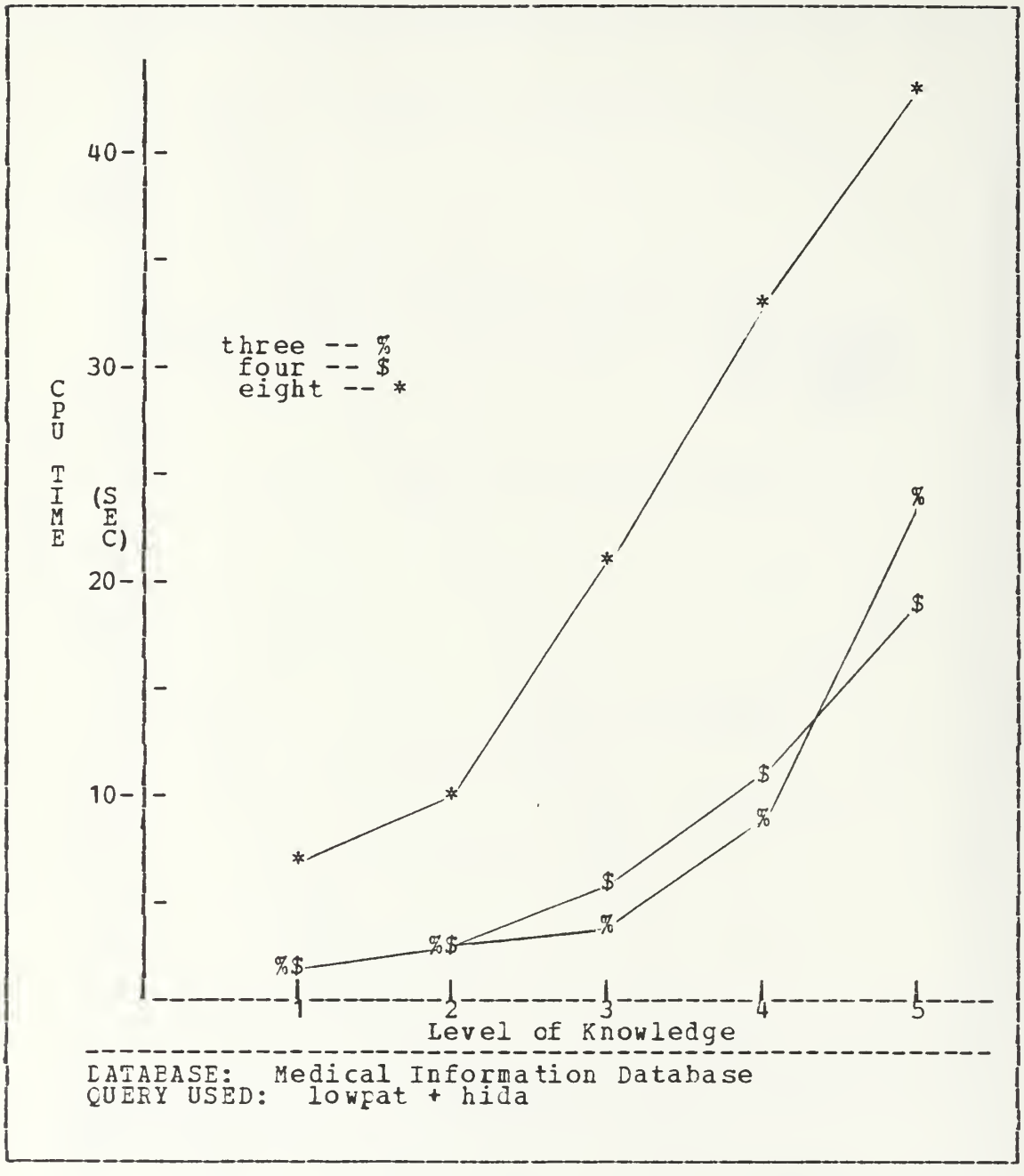

Figure 3.15 Response Times for Query 6 for the Specialized DBA.

optimum refers to the system with most accurate responses, the shortest response times and requiring the least amount of primary storage for the database abstract. Of course,

all of these goals cannot be realized at the same time because they conflict with each other. Therefore tradeoffs must be made to produce the system which best suits the needs of all the users.

First and foremost, it is very important that the database administrator consider which attributes and sets are most commonly queried in the application field. If one particular attribute will be involved in a large percentage of the system queries, that attribute should be partitioned more finely than the other attributes. The question of exactly how finely the partitioning should be has not been determined through these tests. However it is suspected that there exists a granularity for each attribute which is optimum in the sense that partitioning the database more finely on this attribute results in a sharp drop in the improvement in accuracy achieved while the increase in system response time at least remains constant. It is also believed that this optimum granularity is a function of the ratio of the mode frequency of the attribute in the database to the size of the database. For example, a highly uneven distribution in which over half of the tuples of the database have the mode value will have a much lower optimum granularity than a distribution in which the values of the attribute are evenly distributed over the range of values of the attribute. Of course, the number of distinct values in the database for each attribute is an upper bound on the optimum granularity for that attribute because if each partition contained tuples with only one value for the given attribute, any finer partitioning on that attribute would not be possible.

Another way which the database administrator should be familiar with the application field is to identify some partitions of database attributes which are meaningful in the application field and therefore are likely to be used

often in er queries. The example used in chapter I was that of e low, normal and high temperature ranges. If such meaningful partitions are identified, the database administrator can define the database abstract partitions to correspond to the application area partitions of the attribute and result in greatly improved accuracy and system response time.

The optimum granularity of the database abstract, therefore depends on the needs of the user. If the database administrator must provide a system to serve the needs of a variety of users, he can use the normal, alternate and special database abstracts to provide one general purpose and two specialized database abstracts. If this flexibility is still not sufficient, the database administrator must make tradeoffs between the needs of his various users to try to best serve the entire user ccmmunity.

The question of the level of knowledge which must be used to obtain the optimum results for a particular query was determined not to be as important as the granularity of the database abstract. Because the user, rather than the database administrator chooses the level of knowledge to be used for each query, such a decision can easily be altered if the results obtained are not sufficiently accurate. However, the tests conducted indicated that either level one or level five were the optimum levels to use because these levels were capable of calculating new upper and lower bounds on both union and intersection sets.

F. EFFECTS OF USING PROLOG TO IMPLEMENT THE SYSTEM

It was mentioned in chapter II that PROLOG was chosen as the implementation language for this system primarily because it was expected that this language would provide a rule base which was more logical and easily understood than

that which a Lisp system might provide. This advantage was realized to even a greater extent than anticipated, but additional equally important advantages as well as several disadvantages to using PROLOG were also realized. Because this author's experience with Lisp is quite limited, we will concentrate here on the characteristics of PROLOG rather than attempting to compare the two languages.

The syntax of PROLOG (with some exceptions, e.g. ! and =..) is extremely natural for a rule based system. Rules can be constructed individually and translated into PROLOG clauses independent of the other system rules. The form of a PROLOG clause lends itself to working on one level of abstraction at a time using lower level calls to rules whose effect is envisioned but whose implementation has been put off until later. The primary advantage to the PROLOG syntax is the way it lends itself to organizing a program into several files of rules, each of which is understandable independent of the others. For example, a reader could examine the 'infs' file and understand the rules for obtaining lower bounds on the input sets without ever knowing that the 'sups' or 'ests' files exist.

On the other hand, it is quite possible that a reader would need to refer to the 'master' file to determine the effect of many low level functions which are used by the 'infs' file. This leads us to one of the disadvantages of the PROLOG system, that is the lack of many built in functions whose effect is already well understood by the user community. For example, the system does not have a 'member' function to check for the membership of an individual element in a list or an 'append' function to append two lists together. Even the most basic Lisp systems have such built in functions. On the other hand, it is quite easy to write these functions in PROLOG and, once written, they can be used over and over again in various systems. So, except

for the penalties of lower system efficiency, we do not see this disadvantage as too great a problem.

In [Ref. 1], that author spent a great deal of his system's execution time determining which rules in his rule base to apply and ensuring that all the appropriate rules were in fact used for every query. These problems were a direct result of using Lisp as his programming language. PROLOG solves this problem automatically through its pattern matching mechanisms. By providing a sufficient number of arguments in the upper level goals, the PROLOG system can ensure that when the appropriate values are substituted into these goals, only the clause heads of the desired rules will match the goal. Of those rules that match, only the first successful rule will affect the system variables. Therefore, the programmer can dictate the priority of execution of his rules by merely placing the rules in the proper order in the rule base. PROLOG also provides built in features such as the 'cut' predicate to alter the normal search through the rule base whereas in other programming languages, such mechanisms would have to be built from primitives.

Another important way in which PROLOG lends itself to implementing a rule-based system is in its automatic management of the internal PROLOG database of rules and facts. The system in this paper is constructed as if two separate storage areas existed for the original database abstract and the system generated cache. In fact, these rules and facts are all stored in the PROLOG heap. New items can be added to the heap with the 'assert' command and unwanted facts or rules can be removed using the 'retract' and 'abolish' commands. In this way, converting a set union into a simple set with respect to a given level is just a matter of asserting a new set of facts into the internal database. These mechanisms to automatically manage the internal

database eliminate many of the details which would have to be worked out to build such a system in another programming language.

Chapter IV discusses in some detail the possible extensions to this system which could be effected to increase its functionality. These extensions are made possible primarily because of the independent execution of the individual PROLOG rules when these rules have distinct clause heads. A detailed explanation of this idea is deferred until chapter IV but suffice it to say here that this rule-based system could easily be extended to answer queries on statistics other than 'size' and on sets other than unions and intersections. This flexibility is caused primarily by the structure inherent in a PROLOG rule base.

A final advantage of using PROLOG to implement this rule-based system may not be fully realized for several years. With no conscience effort to do so, this system is built to be executed very efficiently on a network of parallel processors. For example, if a parallel conjunction were used between the subgoals 'dosup', 'doinf', and 'doest', these three independent calculations could be carried out in parallel and thus system response time could be reduced considerably. Even more dynamic would be the use of such a parallel conjunction in the 'map' functional to cause parallel execution of the individual rules for all the attributes simultaneously on different processors. These two simple changes in the system would reduce system response time by about an order of magnitude. If a concentrated effort were made from the start of system construction, perhaps even greater improvements in response time could be realized.

With all the above points considered, the decision to use PROLOG as the implementation language for this rule-based system appears to be a wise one. For a penalty of

slightly longer response times and more work to build basic functions, the PROLOG system provided the advantages of logical and easily understood syntax, automatic pattern matching mechanisms for rule application, easy-to-use tools for management of the internal database of rules, and a system which can easily be expanded to increase its functionality. Both of the disadvantages of PROLOG stated above are likely to be overcome in the near future when more widespread use of this language will bring about a larger library of built-in system functions and more research into implementing PROLOG in parallel architectures will dramatically improve system response times.

IV. POSSIBLE EXTENSIONS/IMPROVEMENTS TO THE SYSTEM

A. EXTENDING THE SYSTEM TO ACCEPT QUERIES ON OTHER STATISTICS

The DAQUES system was written with many seemingly unnecessary arguments being passed from clause to clause in the higher level rules. For example, in all three 'do' rules, only one argument is currently needed by the system, that is the set being queried. However, the rules also have places for passing statistic and attribute arguments. Currently, only rules for the 'size' statistic exist in the database of rules. All these rules have clause heads which appear like the following:

```
dosup(size,and(Set_List),_,Answer)
                or
dosup(size,or(Set_List),_,Answer)
```

Here, the third argument would contain the attribute of the statistic to be queried. Since the 'size' statistic is not associated with any particular attribute, this argument remains anonymous in the current system.

In order to extend this system to accept queries on other statistics of the input set, the programmer would simply have to add a new file of rules to supplement the 'sups', 'infs', and 'ests' files. For example, to allow the system to request upper and lower bounds and estimates on the mean value of a particular attribute in the queried set, the programmer would simply have to add rules with clause heads of the form:

```
dosup(mean,and(Set_List),Attribute,Answer)
                and
doinf(mean,and(Set_List),Attribute,Answer)
```

and

doest(mean,and(Set_List),Attribute,Answer)

Similar rules would have to be added for set unions. These new rules would in no way interfere with the current system. No hidden side effects from these new rules could possibly alter the execution of the current rules because the call to the appropriate 'do' rule would only match the rule in the rule base which had the proper statistic as its first argument. Of course, these new rules could use many of the already written general rules which are located in the 'master' file, thus saving the programmer much wasted effort.

It is also anticipated that little or no degradation in system response time would be experienced by augmenting the system in this way. Once the initial invocation of the 'do' clause chose the proper rule to use for the statistic being queried, the rules for the other statistics should have no effect on the speed of execution. Of course, the added rules would increase the space needed to hold the system in the PROLOG internal database. If space became a problem, the programmer could sacrifice response time for added space by only consulting the rules for the specified statistic and then retracting these rules from the rule-base and asserting a different set of rules if a different statistic is queried.

This system could be extended to accept queries on any statistic provided that the appropriate rules could be devised to obtain useful bounds and estimates on those statistics. Additionally, the system could be expanded to accept sets other than the unions and intersections which it currently accepts. For example rules with the following forms could be added to allow the system to calculate upper bounds on sets expressed as complements of simple sets:

dosup(size,comp(Bigset,Set_List),_,Answer)

and

dosup(std_dev,comp(Bigset,Set_List),Attribute,Answer)

Here the notation signifies that the desired query set is made of all tuples from the Bigset which are not also members of any of the sets in the Set_List.

The important point of this section is not to offer specific suggestions as to how to extend the current system but to demonstrate how the flexibility of the current system lends itself to extension. A large part of this flexibility is made possible by the pattern matching ability of the PROLOG language. To construct such a prototype system with the same capability for expansion in a conventional imperative programming language would be practically impossible.

B. IMPROVEMENTS IN SYSTEM ACCURACY

It was discussed in chapter II that a significant improvement in system accuracy could be achieved if the rules for determining upper and lower bounds on the levels two through four information for set unions could be improved. Remember that whenever a set is processed by a 'do' rule, the rule must make that set into a simple set with respect to the current level of knowledge by asserting into the internal database upper and lower bounds on the same level information which is stored about the actual partition sets. After a set union is processed by the level two rules, the upper bound on the mode frequency and number of distinct values which has been asserted into the PROLOG internal database is the sum of these statistics for each of the component sets in the union. This is a terrible bound on these statistics and when this union set is a component set in some higher level union or intersection, the inaccuracy of these bounds is passed on to the calculation of the bounds on the size of the higher level set. Because of the method for converting input sets specified by a range of

values of an attribute into disjoint union of partition sets of the attribute, a new tighter bound on the calculation of the lower level statistics would significantly improve the accuracy of the bounds calculated for the size of the higher level set.

Two general methods for improving the accuracy of the bounds on these level information statistics seem promising. The first is to test the lower level union to detect disjointness and to handle such disjoint sets differently than sets which are not disjoint. The problem with this approach is that it only shows an improvement for the partitioning attribute's calculations. For the other attributes, determining that the sets have no tuples in common does not guarantee that they do not have any values of the non-partitioning attribute in common. For example, in the sample medical database, if we were dealing with the two partition sets patno(1) and patno(2), we could state with confidence that the mode frequency patno of their union was the maximum of the mode frequencies for patno in the two component sets because these sets clearly are disjoint. However, consider the cholesterol attribute. The mode value for cholesterol in both of these sets is the same even though the sets are disjoint. Therefore, if we used the same rule for disjoint sets, the calculated upper bound on the mode frequency of the union would be much less than the actual mode frequency.

The root of the problem in the above paragraph is the fact that for the non-partitioning attribute in the component sets of a union or intersection, we cannot determine whether the mode values of the sets are the same when we are only provided with the mode frequency. The second approach to improving accuracy of these calculations is to augment the database abstract with the actual mode value in addition to the other statistics. This approach would increase the

size of the database abstract by about one third but probably would increase the information content of the database abstract by much more.

C. IMPROVEMENTS IN SYSTEM EFFICIENCY

As stated in chapter I, speed of execution of the system was not a primary concern during its construction. The purpose of a prototype system is not to produce the most efficient system but to prove the feasibility of the approach. Consequently, there is much room for improvement in the efficiency of this system. First of all, the expertise of the author in developing tight PROLOG code (if such a thing exists), matured a great deal from the early to the late stages of the program. Many rules exist in the program, in which a cut predicate might greatly improve the speed of execution of the system. These cut predicates have been inserted in many rules but no determined effort has been made to optimize the code in this way.

Another area which could increase the efficiency of the program is a shift in the overall structure of the highest level predicates. As stated many times previously, part of the purpose of every 'do' rule is to produce the side effect of converting the input set into a simple set with respect to the current level of knowledge in case this set is a component set in a higher level union or intersection. This system inherently wastes a great deal of execution time calculating upper and lower bounds on levels two through five statistics for the highest level sets.

One solution to this wastefulness would be to provide different sets of rules to deal with the highest level sets and the lower level sets. For example, the 'dosup2low' rule would calculate the upper bound on the input set as well as upper bounds on the mode frequency and number of distinct

values of the input set because it is certain that this set is part of a higher level union or intersection. However, the 'dosup2hi' rule would only calculate the upper bounds on the size of the input set because this rule is only called for the highest level sets. These rules could share many of the lower level rules which they call in order to minimize the increase in the size of the rule base for this revised system.

D. SUMMARY

The DAQUES system is designed primarily to demonstrate how the user of a database can be provided with a hierarchy of knowledge to produce increasingly more accurate bounds and estimations for his queries. A primary goal of the system is to provide the most rapid response to users' queries possible, with the most accurate results possible when a user may not require exact answers to his queries or may not have access to the actual database containing that information. The improvements to the basic DAQUES system discussed in this chapter will help the system to better achieve these response time and accuracy goals.

APPENDIX A

SOURCE LISTING FOR THE DATABASE ABSTRACT GENERATION SUBSYSTEM

```
 :- [freqinfo].
 :- [attrlist].
/*************************************************************************/
/*************************    CREATE    ******************************/
/*************************************************************************/
/*  THIS FILE CONTROLS THE USER INTERFACE FOR THE DATABASE ADMINIS-      */
/*  TRATOR TO CREATE THE DESIRED DATABASE ABSTRACT.  HE MUST SPECIFY     */
/*  THE DATABASE TO BE PROCESSED, THE NUMBER OF PARTITIONS FOR EACH      */
/*  ATTRIBUTE IN THE DATABASE, THE LEVEL OF KNOWLEDGE TO BE USED FOR     */
/*  THE CURRENT CALCULATION, AND THE OUTPUT FILE OR FILES TO RECEIVE     */
/*  THE DATABASE ABSTRACT.  THE EXECUTE CLAUSE STARTS THE PROGRAM        */
/*  EXECUTION AND CONTROLS THE FLOW OF INFORMATION FROM THE USER TO      */
/*  THE PROGRAM.                                                         */
/*-----------------------------------------------------------------------*/
 execute :- write('Enter name of database file.'),nl,
            read(Inp),data(attributes,List_of_attrs),
            inpparts(List_of_attrs,Attrno),
            write('Enter level of knowledge-- 1 to 5.'),
            nl,read(I),write('Do you want the standard output file '),
            write('for level '),write(I),
            nl,read(Ans2),test2(Ans2,I,Inp,Attrno),statistics,
            consult('wipeout'),consult('create').

/*************************************************************************/
/*  THE INPARTS CLAUSE INPUTS THE NUMBER OF PARTITIONS FOR EACH ATTRI-   */
/*  FROM THE USER, DETERMINES IF THAT PARTITIONS BOUNDRIES HAVE AL-      */
/*  READY BEEN CALCULATED PREVIOUSLY BY CALLING THE NEEDPARTITION        */
/*  CLAUSE.  THE NEEDPARTITION CLAUSE SEARCHES THE ATTRLIST FILE FOR     */
/*  AN APPROPRIATE PARTITION BOUNDARY LIST AND IF IT CANNOT FIND ONE,    */
/*  IT GIVES THE DATABASE ADMINISTRATOR THE OPTION OF LETTING THE        */
/*  SYSTEM CALCULATE THE BOUNDARIES OR SPECIFYING THEM HIMSELF.  IF      */
/*  HE CHOOSES TO SPECIFY THE BOUNDARIES, HE MAY ESTABLISH ALIASES       */
/*  FOR THE PARTITION NAMES OF ONE OR MORE OF THE ATTRBUTES.             */
/*-----------------------------------------------------------------------*/
 inpparts([ ],[ ]).
 inpparts([Attr|Allattrs],[[Attr,Pno]|Attrparts]) :- non_numeric(Attr,Pno),!,
            write(Attr),write(' is a non-numeric attribute and can only'),nl,
            write('be partitioned into '),write(Pno),write(' partitions.'),nl,
            inpparts(Allattrs,Attrparts).
 inpparts([Attr|Allattrs],[[Attr,Pno]|Attrparts]) :-
            write('Enter number of partitions for the '),write(Attr),
```

```
        write(' attribute.'),nl,read(Pno),needpartition(Pno,Attr),
        inpparts(Allattrs,Attrparts).

/*****************************************************************/
/*   THE NEEDPARTITION CLAUSE CHECKS TO SEE IF A GIVEN PARTITION SIZE */
/*   HAS ITS BOUNDRIES ALREADY TABULATED IN THE ATTRLIST FILE. IF SO  */
/*   THE CLAUSE SUCCEEDS.  IF NOT THEN THE PARTITION CLAUSE IS CALLED */
/*   TO CALCULATE SUCH BOUNDRIES.  THE PARTITION CLAUSE ACCESSES THE  */
/*   FREQUENCY DISTRIBUTION FOR A PARTICULAR ATTRIBUTE IN THE UNIVERSE*/
/*   AND USES THIS INFORMATION TO CALCULATE THE DESIRED NUMBER OF     */
/*   PARTITIONS FOR THE ATTRIBUTE.  THE GOAL IS TO HAVE EACH PARTITION*/
/*   CONTAIN APPROXIMATELY THE SAME NUMBER OF VALUES SO THE SIZE OF   */
/*   THE UNIVERSE IS DIVIDED BY THE NUMBER OF PARTITIONS DESIRED TO   */
/*   DETERMINE THE SIZE OF EACH PARTITION SET.  HOWEVER BECAUSE ALL   */
/*   VALUES WHICH ARE THE SAME MUST BE INCLUDED IN THE SAME PARTITION,*/
/*   THE FIRST PARTITION MAY CONTAIN MORE THAN THE PREVIOUSLY CALCU-  */
/*   LATED NUMBER OF ITEMS.  THEREFORE, THE SIZE OF THE REMAINING     */
/*   PARTITIONS IS CALCULATED BASED ON THE SIZE OF THE UNIVERSE MINUS */
/*   THE NUMBER OF VALUES INCLUDED IN THE FIRST PARTITION.  BECAUSE   */
/*   THIS METHOD TENDS TO GROUP THE MAJORITY OF VALUES INTO THE EARLY */
/*   PARTITIONS, AN ACTUAL VALUE OF NINE TENTHS OF THE CALCULATED SIZE*/
/*   OF THE NEXT PARTITION IS USED TO DETERMINE THE PARTITION BOUN-   */
/*   DARY.                                                           */
/*-----------------------------------------------------------------*/
needpartition(Noparts,Attr) :- attributelist(Attr,Noparts,_).
needpartition(Noparts,Attr) :- not(attributelist(Attr,Noparts,_)),
            write('Do you wish to explicitly specify partition boundaries?'),
            nl,read(Ans),qspecify(Attr,Noparts,Ans,Alist),tell(attr),
            writeq(attributelist(Attr,Noparts,Alist)),write('.'),nl,told,
            system("cat attr attrlist>temp"),system("cp temp attrlist"),
            system("rm temp attr"),asserta(attributelist(Attr,Noparts,Alist)).
qspecify(Attr,Noparts,no,Alist) :- partition(Attr,Noparts,Alist).
qspecify(Attr,Noparts,yes,Alist) :- readboundaries(Noparts,Alist),
             write('Do you wish to alias these specified partitions?'),
             nl,read(Ans),qalias(Ans,Attr,Noparts).
qalias(no,_,_).
qalias(yes,Attr,Noparts) :- write('Enter a list of '),write(Noparts),nl,
            write(' alias names for the partitions you have specified.'),
            nl,read(Aliaslist),tell(tmpalias),
            writealias(Attr,Noparts,1,Aliaslist),told,
            system("cat tmpalias alias >temp"),system("cp temp alias"),
            system("rm temp").
writealias(Attr,Noparts,Noparts,[Lastalias]) :- groupname(Attr,Noparts,Name),
            writeq(alias(Noparts,Lastalias,Name)),write('.'),nl.
writealias(Attr,Noparts,Num,[Nextalias|Rest]) :- groupname(Attr,Num,Name),
            writeq(alias(Noparts,Nextalias,Name)),write('.'),nl,
            Newnum is Num + 1,writealias(Attr,Noparts,Newnum,Rest).
readboundaries(1,[]).
```

```
readboundaries(No,Alist) :-  Num is No - 1,
            write('Enter a list of '),write(Num),write(' numbers.'),nl,
       write('Each number is tc be an upper inclusive bound for a partition.'),

       nl,read(Alist).
partition(Value,Noparts,N) :- data(freqdist,univ,Value,L),
                              part(499,Noparts,L,N,0).
part(Totleft,1,L,[ ],0).
part(Totleft,Partsleft,[[X,Y]|L],N,Sofar) :-
                       Fsize is Totleft/Partsleft,Subt is Fsize/10,
                       Size is Fsize-Subt,A is Sofar+Y,A<Size,
                       part(Totleft,Partsleft,L,N,A).
part(Totleft,Partsleft,[[X,Y]|L],[X|N],Sofar) :-
                       Fsize is Totleft/Partsleft,Subt is Fsize/10,
                       Size is Fsize-Subt,A is Sofar+Y,A>=Size,
                       Ntot is Totleft-A,Nparts is Partsleft-1,
                       part(Ntot,Nparts,L,N,0).

/*****************************************************************/
/*  THE TEST2 CLAUSE ENSURES THAT THE PROPER OUTPUT FILE OR FILES  */
/*  ARE SET UP TO RECEIVE THE DATABASE ABSTRACT FILES AND THEN CALLS*/
/*  THE EX CLAUSE WHICH SERVES AS A LINK TO THE PROPER FILE TO     */
/*  DO THE ACTUAL CALCULATION OF THE DATABASE ABSTRACT.            */
/*----------------------------------------------------------------*/
test2(yes,I,Inp,Attrno) :- consulti(I,Outp),ex(Inp,Outp,I,Attrno).
test2(no,I,Inp,Attrno) :- write(' Please enter name of output file(s).'),nl,
            read(Outp),consulti(I,_),ex(Inp,Outp,I,Attrno).
ex(Inp,Outp,1,Attrno) :- exec(Inp,Outp,all,Attrno),
            write('Database abstract for level '),write('1'),
            write(' can be found in '),write(Outp),nl.
ex(Inp,Outp,I,Attrno) :- integer(I),I>1,I=<5,exec(Inp,Outp,all,all,Attrno),
            write('Database abstract for level '),write(I),
            write(' can be found in '),write(Outp),nl.
ex(Inp,Outp,all,Attrno) :- exec(Inp,Outp,all,all,Attrno),
            write('Database abstract for levels 2 thru 5 can be found in '),nl,
            write(Outp).
ex(Inp,Outp,I,_) :-  (I>5;I=<0),write('You must enter a level from 1 to 5.'),
                  nl,write('Please try again!').

/*****************************************************************/
/*  THE CONSULTI CLAUSE ENSURES THAT THE CONTINUE CLAUSE WHICH IS  */
/*  CALLED IN THE FREQUENCIES FILE CORRESPONDS TO THE CORRECT LEVEL*/
/*  OF KNOWLEDGE I.E. THAT REQUESTED BY THE USER.                  */
/*----------------------------------------------------------------*/
consulti(1,fact1) :- consult('level1').
consulti(2,fact2) :- consult('level2').
consulti(3,fact3) :- consult('level3').
consulti(4,fact4) :- consult('level4').
```

```
consulti(5,fact5) :- consult('level5').
consulti(all,[fact2,fact3,fact4,fact5]) :- consult('alllevels').
consulti(I,_) :- I>5.
consulti(I,_) :- I=<0.
```

```
:- ([qsort,find]).
/*************************************************************************/
/*************************   FREQUENCIES   ***************************/
/*************************************************************************/
/*   THIS FILE IS USED FOR LEVELS TWO, THREE, FOUR AND FIVE KNOWLEDGE.*/
/*   IT READS IN THE ENTIRE DATABASE AND BUILDS A LIST FOR EACH PAR-  */
/*   TITION SET FOR EACH ATTRIBUE.  THE LISTS CONTAIN TWO ELEMENT     */
/*   LISTS.  THE FIRST ELEMENTS ARE THE VARIOUS VALUES FOR THE ATTRI- */
/*   BUTES FOUND IN THE PARTICULAR PARTITION SET AND THE SECOND ELE-  */
/*   MENT IS THE CORRESPONDING FREQUENCY OF OCCURRANCE OF THAT VALUE  */
/*   IN THAT SET.  THESE LISTS CORRESPOND TO LEVEL 5 INFORMATION BUT  */
/*   CAN BE USED EASILY TO DETERMINE LEVELS 2, 3, AND 4 INFORMATION   */
/*   ALSO.                                                           */
/*-------------------------------------------------------------------*/

/*************************************************************************/
/*   THE EXEC CLAUSE IS THE LINK TO THE CREATE FILE.  THE USER INTER- */
/*   FACE PASSES THE FREQUENCIES FILE THE NAME OF THE DATABASE FILE,  */
/*   THE NAME OF THE OUTPUT FILE OR FILES FOR THE DATABASE ABSTRACT,  */
/*   AND THE LIST OF ATTRIBUTES AND THE NUMBER OF PARTITIONS FOR EACH.*/
/*   THE THIRD AND FOURTH AGRUMENTS EXIST SO THAT THIS FILE COULD BE  */
/*   USED TO BUILD ONLY A SUBSET OF THE DATABASE ABSTRACT.  THE FIRST */
/*   EXEC CLAUSE DEALS WITH A REQUEST TO BUILD A DATABASE ABSTRACT    */
/*   A SPECIFIED LEVEL OF KNOWLEDGE WHILE THE SECOND HANDLES THE CASE */
/*   WHEN THE USER WANTS THE DATABASE ABSTRACT BUILT FOR ALL LEVELS.  */
/*   THE ASSERTNAMES CLAUSE ASSERTS THE NAMES OF THE OUTPUT FILES WHEN*/
/*   ALL LEVELS ARE REQUESTED.  THIS INFORMATION IS USED LATER BY THE */
/*   CONTINUE CLAUSES.  THE CALCFREQS CLAUSE CONTROLS THE FLOW OF     */
/*   CONTROL THROUGH THE PROGRAM.  FIRST THE NECESSARY INITIAL CONDI- */
/*   TIONS ARE ASSERTED WITH TOTALASSERT THEN THE LISTS ARE CALCULATED*/
/*   WITH THE CALC CLAUSE.  FINALLY THE LISTS ARE PROCESSED ACCORDING */
/*   TO THE REQUESTED LEVEL OF KNOWLEDGE BY THE CONTINUE CLAUSES.     */
/*-------------------------------------------------------------------*/
  exec(Inp,Outp,Attr,Value,Attrno) :- atomic(Outp),tell(Outp),see(Inp),
                                      calcfreqs(Attr,Value,Attrno),seen,told.
  exec(Inp,Outp,Attr,Value,Attrno) :- assertnames(2,Outp),see(Inp),
                                      calcfreqs(Attr,Value,Attrno),seen,told.

  assertnames(_,[]).
  assertnames(Num,[Name|Rest]) :- asserta(filename(Num,Name)),Newnum is Num+1,
                                  assertnames(Newnum,Rest).
  calcfreqs(all,all,Attrno) :- totalassert(Attrno,Attrno),
                               calc(all,all,Attrno),totcontinue(Attrno,Attrno),!.

/*************************************************************************/
/*   THE ASSERT CLAUSES ASSERT EMPTY LISTS FOR EVERY PARTITION SET OF EVERY */
/*   ATTRIBUTE WITH RESPECT TO EVERY ATTRIBUTE.  THE TOTALASSERT CLAUSE     */
```

```
/*  IS THE HIGHEST LEVEL CLAUSE AND RUNS THROUGH THE LIST OF ATTRIBUTES    */
/*  CALLING ASSERTALLATTRS FOR EACH ATTRIBUTE.  THE ASSERTALLATTRS RUNS    */
/*  THROUGH EACH ATTRIBUTE IN THE ATTRIBUTE LIST AGAIN THIS TIME CALLING   */
/*  ASSERTALL FOR EACH ATTRIBUTE.  IN THIS WAY ASSERTALL IS CALLED FOR     */
/*  EVERY MEMBER OF THE CARTESIAN PRODUCT OF THE SET OF ATTRIBUTES WITH    */
/*  ITSELF.  THE ASSERTALL CLAUSE RUNS THROUGH THE PARTITIONS FOR THE FIRST*/
/*  VALUE FOR ATTRIBUTE WITH RESPECT OT THE SECOND VALUE FOT ATTRIBUTE.    */
/*  THIS CLAUSE USES THE NUMBER OF PARTITIONS FOR EACH ATTRIBUTE WHICH WAS */
/*  INPUT FROM THE USER.                                                   */
/*-------------------------------------------------------------------------*/
  totalassert([ ], _).
  totalassert([[Attr,Attrno]|Rest],Alist) :- assertallattrs(Attr,Attrno,Alist),
                                             totalassert(Rest,Alist).
  assertallattrs(_,_,[ ]).
  assertallattrs(Attr,Attrno,[[sex, _]|Rest]) :-
                                             assertallattrs(Attr,Attrno,Rest).
  assertallattrs(Attr,Attrno,[[A,_]|Rest]) :- assertall(Attr,A,Attrno,1),
                                             assertallattrs(Attr,Attrno,Rest).
  assertall(Attr,Value,Attrno,Attrno) :- groupname(Attr,Attrno,Gpname),
                                         asserta(data(freqdist,Gpname,Value,[ ])).
  assertall(Attr,Value,Attrno,Group) :- groupname(Attr,Group,Gpname),
                                        asserta(data(freqdist,Gpname,Value,[ ])),
                                        B is Group+1,assertall(Attr,Value,Attrno,B).

/*************************************************************************/
/*  THESE RULES PERFORM THE MAJORITY OF THE WORK OF THIS SYSTEM.  THE     */
/*  READALL CLAUSE CONTROLS THE INPUT OF ALL OF THE TUPLES FROM THE       */
/*  DATABASE.  THE TOTCHECK CLAUSE DOES THE ACTUAL WORK ON EACH TUPLE     */
/*  READ IN.  THE CUT PREDICATE AT THE END OF THE TOTCHECK CLAUSE IS      */
/*  VERY IMPORTANT BECAUSE EACH TIME THAT THE A=='end of file' CHECK      */
/*  FAILS, THE SYSTEM MUST BACKTRACK DIRECTLY TO THE REPEAT CLAUSE IN     */
/*  ORDER TO READ IN THE NEXT TUPLE.  THIS CUT PREDICATE PREVENTS THE     */
/*  SYSTEM FROM ATTEMPTING TO RESATISFY ANY LOWER LEVEL CLAUSES UNDER     */
/*  THE TOTCHECK RULE DURING THIS BACKTRACKING.  THE TOTALCHECK CLAUSE    */
/*  DETERMINES WHAT PARTITION THE CURRENT TUPLE BELONGS TO WITH RESPECT   */
/*  TO THE CURRENT ATTRIBUTE.  IT THEN CALLS CHECKALL FOR EACH NON-       */
/*  NUMERIC ATTRIBUTE AND PASSES THE VALUE OF THE GROUP FOR THAT TUPLE    */
/*  AND A CURRENT ATTRIBUTE.  THE CHECKALL CLAUSE CALLS CHECK TO FIND     */
/*  THE VALUE OF THE INNER ATTRIBUTE (DONE BY FIND), THEN IT FINDS THE    */
/*  PREVIOUSLY ASSERTED LIST FOR THAT PARTITION SET AND THE INNER ATTRI-  */
/*  BUTE.  THE INTOLIST CLAUSE SEARCHES THE LIST FOR THE AFOREMENTIONED   */
/*  VALUE.  IF IT IS FOUND, ITS FREQUENCY IS INCREMENTED BY ONE.  IF      */
/*  NOT, THE VALUE IS ADDED TO THE LIST WITH A FREQUENCY OF ONE.  THE     */
/*  CHECKALL CLAUSE CHECKS THE CURRENT TUPLE FOR ALL OF ITS VALUES AND    */
/*  THEN THE TOTALCHECK CLAUSE MOVES ON AND FINDS THE GROUP OF THE TUPLE  */
/*  WITH RESPECT TO THE NEXT ATTRIBUTE IN THE LIST.                       */
/*-----------------------------------------------------------------------*/
  calc(Attr,Value,Attrno) :- readall(Attr,Value,Attrno).
```

```
readall(all,all,Attrno) :- new_read(A),totcheck(A,Attrno,Attrno),
                           A=='end_of_file',!.
totcheck(A,Attrno,Attrno) :- A=='end_of_file',!.
totcheck(A,Attrno,Attrno) :- totalcheck(A,Attrno,Attrno),!.
totalcheck(A,[ ],Attrno).
totalcheck(A,[[Val,Pno]|Rest],Attrno) :- findstuff(A,Val,Group,Pno),
                        checkall(A,Attrno,Group),
                        totalcheck(A,Rest,Attrno),!.
checkall(A,[ ],Group).
checkall(A,[[sex,_]|Rest],Group) :- checkall(A,Rest,Group).
checkall(A,[[Val,_]|Rest],Group) :- check(A,Group,Val),
                                    checkall(A,Rest,Group).
new_read(A) :- repeat,read(A).
check(A,Group,Value) :- A == 'end_of_file',!.
check(A,Group,Value) :-data(freqdist,Group,Value,L),
                       find(A,P,Value),intolist(P,L,N),
                       retract(data(freqdist,Group,Value,L)),
                       asserta(data(freqdist,Group,Value,N)),!.

intolist(P,[ ],[[P,1]]).
intolist(P,[[P,A]|L],[[P,B]|L]) :- B is A+1,!.
intolist(P,[[A,B]|L],[[A,B]|N]) :- P>A,intolist(P,L,N).
intolist(P,[[A,B]|L],N) :- P<A,addto([[A,B]|L],P,N).
addto(L,P,[[P,1]|L]).

/*****************************************************************/
/*  THE CONTINUE CLAUSES ARE EXACTLY ANALOGOUS TO THE ASSERT CLAUSES  */
/*  IN THE WAY THEY PASS THROUGH THE ATTRIBUTES OF THE DATABASE.  IN- */
/*  STEAD OF ASSERTING A FACT WHEN THE INNER LEVEL IS REACHED THE     */
/*  CONTINUE CLAUSES MAKE A CALL TO THE CONTINUE CLAUSE WHOSE VALUE   */
/*  WILL VARY DEPENDING ON WHICH LEVEL FILE WAS ASSERTED DURING THE   */
/*  USER INTERFACE  WHICH IN TURN DEPENDS ON THE LEVEL OF KNOWLEDGE   */
/*  INPUT BY THE USER.                                               */
/*-------------------------------------------------------------------*/
totcontinue([ ],Attrno).
totcontinue([[Attr,Pno]|Rest],Attrno) :- write('/* DATA FOR THE '),
                        write(Attr), write(' ATTRIBUTE */'),nl,
                        continueallattrs(Attr,Attrno,Pno),
                        totcontinue(Rest,Attrno).
continueallattrs(Attr,[ ],Pno).
continueallattrs(Attr,[[sex,_]|Rest],Pno) :- continueallattrs(Attr,Rest,Pno).
continueallattrs(Attr,[[Val,_]|Rest],Pno) :- continueall(Attr,Val,Pno,1),
                        continueallattrs(Attr,Rest,Pno).
continueall(Attr,Value,Attrno,Attrno) :- groupname(Attr,Attrno,Gpname),
                        data(freqdist,Gpname,Value,L1),
                        continue(L1,Gpname,Value,Attr).
continueall(Attr,Value,Attrno,Group) :- groupname(Attr,Group,Gpname),
                        data(freqdist,Gpname,Value,L1),
```

```
continue(L1,Gpname,Value,Attr),
B is Grcup+1,continueall(Attr,Value,Attrno,B),!.
```

```
:- ([find]).
/*****************************************************************/
/*************************   LEVEL1   ****************************/
/*****************************************************************/
/*  THIS FILE CALCULATES THE LEVEL 1 DATABASE ABSTRACT WHEN THAT    */
/*  IS REQUESTED BY THE USER.  THE EXEC CLAUSE IS THE LINK BETWEEN  */
/*  THIS FILE AND THE CREATE USER INTERFACE FILE.  IT RECEIVES THE  */
/*  NAME OF THE DATABASE, THE NAME OF THE FILE TO RECEIVE THE LEVEL */
/*  1 DATABASE ABSTRACT, AND THE LIST OF ATTRIBUTES AND NUMBER OF   */
/*  PARTITIONS FOR EACH REQUESTED BY THE USER.  THE THIRD AND FOURTH*/
/*  ARGUMENTS ALLOW A SUBSET OF THE DATABASE ABSTRACT TO BE CALCU-  */
/*  LATED.  THE EXECUTE CLAUSE ACCESSES THE DESIRED OUTPUT FILE AND */
/*  CALLS THE PRECALC CLAUSE WHICH ASSERTS THE NECESSARY INITIAL    */
/*  CONDITIONS INTO THE INTERNAL DATABASE AND THEN CALL THE CALC    */
/*  CLAUSE TO DO THE BULK OF THE WORK FOR THE PROGRAM.              */
/*-----------------------------------------------------------------*/
exec(Inp,Outp,all,Attrno) :- tell(Outp),write(partitions(Attrno)),
                             write('.'),nl,execute(all,Inp,Attrno),
                             totcontinue(Attrno),
                             tot(nil,univ,X),
                             Y is X-1,writeq(data(size,univ,nil,Y)),
                             write('.'),nl,told,!.
execute(Attr,Inp,Attrno) :- see(Inp),precalc(Attr,Attrno).
precalc(all,Attrno) :- totassert(Attrno),calc(all,Attrno),!.

/*****************************************************************/
/*  THE TOTALASSERT CLAUSE RUNS THROUGH ALL THE ATTRIBUTES IN THE  */
/*  ATTRNO LIST AND CALLS ASSERTALL FOR EACH.  THE ASSERTALL CLAUSE*/
/*  ASSERTS THE INITIAL VALUE OF ZERO FOR THE SIZE OF EACH PARTI-  */
/*  TION SET FOR THE GIVEN ATTRIBUTE.  THE NUMBER OF PARTITIONS    */
/*   FOR A GIVEN ATTRIBUTE IS FOUND IN THE ATTRNO LIST.            */
/*-----------------------------------------------------------------*/
totassert([]) :- asserta(tot(nil,univ,0)).
totassert([[Attr,Attrno]|Rest]) :- assertall(Attr,Attrno,1),totassert(Rest).
assertall(Attr,Attrno,Attrno) :- groupname(Attr,Attrno,Gpname),
                                 asserta(tot(Attr,Gpname,0)),
                                 asserta(maxval(Attr,0)),asserta(minval(Attr,1000)).
assertall(Attr,Attrno,Group) :- groupname(Attr,Group,Gpname),
                                asserta(tot(Attr,Gpname,0)),B is Group+1,
                                assertall(Attr,Attrno,B).

/*****************************************************************/
/*   THE CALC CLAUSE CALLS READALL WHICH CONTROLS THE READING AND  */
/*   PROCESSING OF EACH TUPLE IN THE DATABASE.  THE CUT PREDICATE  */
/*   AT THE END OF THE CALCULATE PREDICATE IS VERY IMPORTANT TO    */
/*   ENSURE THAT WHEN THE L=='end_of_file' CONDITION FAILS  THE    */
```

```
/*    BACKTRACKING MECHANISM WILL PROCEDE DIRECTLY BACK TO THE      */
/*    REPEAT CLAUSE TO READ IN THE NEXT TUPLE INSTEAD OF TRYING TO  */
/*    RESATISFY THE LOWER LEVEL GOALS UNDER THE CALCULATE PREDICATE.*/
/*    THE CHECKALL CLAUSE CALLS CHECK FOR EACH ATTIBUTE IN THE      */
/*    ATTRNO LIST.  CHECK CALLS FIND AND FINDGROUP TO DETERMINE THE */
/*    PARTITION SET TO WHICH THE CURRENT TUPLE BELONGS WITH RESPECT */
/*    TO THE GIVEN ATTRIBUTE.  THE SIZE OF THIS GROUP IS THEN FOUND */
/*    IN THE INTERNAL DATABASE NAD INCREMENTED BY 1 USING THE       */
/*    INCR1TOT CLAUSE.   THE CHECK CLAUSE ALSO DETERMINES IF THE    */
/*    CURRENT TUPLES VALUE FOR THE GIVEN ATTRIBUTE IS LESS THAN THE */
/*    CURRENTLY TABULATED MINIMUM VALUE OR GREATER THAN THE CUR-    */
/*    RENTLY TABULATED MAXIMUM VALUE REPLACING THESE VALUES IF      */
/*    NECESSARY.  THE QASSERT CLAUSE ACCOMPLISHHES THIS DECISION.   */
/*--------------------------------------------------------------------*/
  calc(Attr,Num) :- readall(Attr,Num).
  readall(all,Attrno) :- new_read(L),calculate(L,Attrno),L=='end_of_file',!.
  calculate(L,Attrno) :- checkall(L,Attrno),!.
  checkall(L,[]) :- incr1tot(nil,univ),!.
  checkall(L,[[Attr,Attrno]|Rest]) :- check(L,Attr,Attrno),checkall(L,Rest).
  new_read(L) :- repeat,read(L).
  check(L,Attr,Attrno) :- L=='end_of_file',!.
  check(A,Attr,Attrno) :- find(A,S,Attr),findgroup(Attr,S,Group,Attrno),
                          incr1tot(Attr,Group),minval(Attr,Min),
                          qassert(Attr,Min,S,lt),maxval(Attr,Max),
                          qassert(Attr,Max,S,gt).
  qassert(Attr,_,_,_) :- non_numeric(Attr).
  qassert(Attr,M,S,lt) :- M=<S.
  qassert(Attr,M,S,lt) :- M>S,retract(minval(Attr,M)),asserta(minval(Attr,S)).
  qassert(Attr,M,S,gt) :- M>=S.
  qassert(Attr,M,S,gt) :- M<S,retract(maxval(Attr,M)),asserta(maxval(Attr,S)).
  incr1tot(Set,S) :- tot(Set,S,T),retract(tot(Set,S,T)),A is T+1,
                     asserta(tot(Set,S,A)).

/*********************************************************************/
/*  THE TOTCONTINUE CLAUSE IS CALLED AFTER THE ENTIRE DATABASE HAS   */
/*  BEEN PROCESSED.  IT RUNS THROUGH THE ATTRIBUTES IN THE ATTRNO    */
/*  LIST AND WRITES THE TOTALS CALCULATED ABOVE TO THE DESIGNATED    */
/*  OUTPUT FILE.                                                     */
/*-------------------------------------------------------------------*/
  totcontinue([]).
  totcontinue([[Attr,Attrno]|Rest]) :- continue(Attr,Attrno,1),
                                       totcontinue(Rest).
  continue(Attr,Attrno,Attrno) :- groupname(Attr,Attrno,Gpname),
                                  tot(Attr,Gpname,X1),
                                  writeq(data(size,Gpname,Attr,X1)),write('.'),nl,
                                  minval(Attr,Min),writeq(minval(Attr,Min)),write('.'),nl,
                                  maxval(Attr,Max),writeq(maxval(Attr,Max)),write('.'),nl.
  continue(Attr,Attrno,Group) :- groupname(Attr,Group,Gpname),
```

```
tot(Attr,Gpname,X1),
writeq(data(size,Gpname,Attr,X1)),write('.'),nl,
B is Group+1,continue(Attr,Attrno,B).
```

```
/*************************************************************************/
/*************************    LEVEL2   ***********************************/
/*************************************************************************/
/*  THIS FILE CONTAINS THE CONTINUE CLAUSE FOR A LEVEL 2 DATABASE AB-    */
/*  STRACT.  IT IS CONSULTED WHEN THE USER REQUESTS A LEVEL 2 DATABASE   */
/*  ABSTRACT BE CREATED.  THE INPUT LIST IS A KEYED LIST OF ITEMS AND    */
/*  FREQUENCIES IN INCREASING ORDER OF THE ITEM.  THE QSORT CLAUSE       */
/*  SORTS THE LIST INTO DESCENDING ORDER OF THE FREQUENCIES.  THUS THE   */
/*  FIRST ELEMENT REPRESENTS THE FREQUENCY OF THE MOST FREQUENT ITEM     */
/*  AND THE LENGTH OF THE FREQUENCY LIST REPRESENTS THE NUMBER OF DIS-   */
/*  TINCT ITEMS IN THE SET.  THESE VALUES ARE OUTPUT TO THE FILE DESIG-  */
/*  NATED TO RECEIVE THE LEVEL2 DATABASE ABSTRACT.                       */
/*-----------------------------------------------------------------------*/
  :- ([frequencies]).
  continue(L,A,Value,Attr) :- qsort(L,Sl),length(Sl,Nodist),
                              ith(Sl,1,[Mode,Modf]),
                              writeq(data(modefreq,A,Value,Modf)),write('.'),nl,
                              writeq(data(nodist,A,Value,Nodist)),write('.'),nl.

/*************************************************************************/
/*************************    LEVEL3   ***********************************/
/*************************************************************************/
/*  THIS FILE IS CONSULTED WHEN THE USER REQUESTS LEVEL 3 KNOWLEDGE AND  */
/*  PROVIDES THE CONTINUE CLAUSE TO CALCULATE THAT LEVEL OF KNOWLEDGE.   */
/*  THE CLAUSE INPUTS A KEYED LIST OF ITEMS AND THEIR FREQUENCIES IN     */
/*  ASCENDING ORDER OF THE ITEM.  THE QSORT CLAUSE REARRANGES THE LIST   */
/*  SO THAT IT IS ORDERED BY DESCENDING ORDER OF THE FREQUENCY.  THEN    */
/*  THE SECOND ITEM IN THE LIST IS THE FREQUENCY OF THE SECOND MOST      */
/*  COMMON ITEM IN THE LIST AND THE n/2ND ITEM IS THE FREQUENCY OF THE   */
/*  MEDIAN FREQUENCY ITEM IN THE LIST IF N = LENGTH OF THE LIST.         */
/*-----------------------------------------------------------------------*/
  :- ([fact1]).
  :- ([frequencies]).
  continue(L,A,Value,Attr) :- qsort(L,Sl),length(Sl,Nodist),
                              ith(Sl,2,[Mode2,Modf2]),
                              Half is Nodist/2,round(Half,Newhalf),
                              I is Newhalf+1,ith(Sl,I,[_,Medf]),
                              writeq(data(modf2,A,Value,Modf2)),write('.'),nl,
                              writeq(data(medf,A,Value,Medf)),write('.'),nl.
  round(0.5,1).
  round(X,X) :- integer(X).
  round(X,Y) :- not(integer(X)),Y is X - 0.5.

/*************************************************************************/
/*************************    LEVEL4   ***********************************/
/*************************************************************************/
```

```
/*  THIS FILE IS CONSULTED WHEN THE USER REQUESTS LEVEL 4 KNOWLEDGE AND */
/*  IT PROVIDES THE CONTINUE CLAUSE FOR CALCULATING THAT LEVEL OF KNOW- */
/*  LEDGE.  THE INPUT IS A KEYED LIST OF ITEMS AND THEIR FREQUENCIES IN */
/*  ASCENDING ORDER OF THE ITEM.  THE QSORT CLAUSE REARRANGES THEM INTO */
/*  DESCENDING ORDER OF THE FREQUENCIES.  THE PULOUTFREQS CLAUSE CREATES*/
/*  A NEW LIST CONTAINING ONLY THE FREQUENCIES FROM THE INPUT LIST STILL*/
/*  IN DESCENDING ORDER.  THIS LIST IS THEN WRITTEN TO THE FILE DESIG-  */
/*  NATED TO RECEIVE THE LEVEL 4 DATABASE ABSTRACT.                     */
/*--------------------------------------------------------------------*/
  :- ([frequencies]).
  continue(L,A,Value,Attr) :- qsort(L,S1),puloutfreqs(S1,L1),
                              writeq(data(freqdist4,A,Value,L1)),
                              write('.'),nl,nl,nl.
  puloutfreqs([ ],[ ]).
  puloutfreqs([[A,B]|L1],[B|L2]) :- puloutfreqs(L1,L2).

/**********************************************************************/
/*************************    LEVEL5    *******************************/
/**********************************************************************/
/*  THIS FILE IS CONSULTED WHEN THE USER REQUESTS TO BUILD A LEVEL 5    */
/*  DATABASE ABSTRACT AND PROVIDES THE CONTINUE CLAUSE TO DO SO.  IN THE*/
/*  CASE OF LEVEL 5 KNOWLEDGE, THE INPUT LIST TO THE CONTINUE CLAUSE IS */
/*  IN FACT EXACTLY THE LEVEL 5 KNOWLEDGE SO THESE LISTS ARE MERELY OUT-*/
/*  PUT TO THE FILE DESIGNATED TO RECEIVE THE LEVEL 5 DATABASE ABSTRACT.*/
/*--------------------------------------------------------------------*/
  :- ([frequencies]).
  continue(L,A,Value,Attr) :- writeq(data(freqdist,A,Value,L)),
                              write('.'),nl,nl,nl.

/**********************************************************************/
/*************************   ALLLEVELS    *******************************/
/**********************************************************************/
/*  THIS FILE IS CONSULTED WHEN THE USER REQUESTS TO CREATE DATABASE    */
/*  ABSTRACTS FOR LEVELS 2 THROUGH 5 ALL AT ONCE.  IN THIS CASE THE     */
/*  CONTINUE CLAUSE PROVIDED CALLS THE OLD CONTINUE CLAUSES FOR ALL     */
/*  FOUR LEVELS OF KNOWLEDGE.  THESE CLAUSES HAVE BEEN RENAMED HERE TO  */
/*  SPECIFY WHICH LEVEL OF KNOWLEDGE THEY CALCULATE BUT THEIR RULES ARE */
/*  EXACTLY THE SAME AS THOSE EXPLAINED IN FILES LEVEL2, LEVEL3, LEVEL4,*/
/*  AND LEVEL5 EXCEPT THAT THE FILENAME CLAUSE IS INSERTED IN ORDER FOR */
/*  EACH CONTINUE CLAUSE TO KNOW WHICH FILE HAS BEEN DESIGNATED TO RE-  */
/*  CEIVE ITS PARTICULAR LEVEL OF KNOWLEDGE.                            */
/*--------------------------------------------------------------------*/

  :- ([frequencies]).
  :- ([fact1]).
  continue(L,Gpname,Value,Attr) :- continue2(L,Gpname,Value,Attr),!,
                                   continue3(L,Gpname,Value,Attr),!,
```

```
                        continue4(L,Gpname,Value,Attr),!,
                        continue5(L,Gpname,Value,Attr).

/*****************************************************************/
  continue2(L,A,Value,Attr) :- filename(2,Name),tell(Name),qsort(L,Sl),
                        length(Sl,Nodist),ith(Sl,1,[Mode,Modf]),
                        writeq(data(modefreq,A,Value,Modf)),write('.'),nl,
                        writeq(data(nodist,A,Value,Nodist)),write('.'),nl.
  try(Set,Attr) :- data(freqdist,Set,Attr,L),continue(L,Set,Attr,_).

  allatts(Value) :- tryall(patno,Value),
                    tryall(da,Value),
                    tryall(temp,Value),tryall(chol,Value),tryall(pred,Value).
  tryall(Attr,Value) :- groupname(Attr,1,Name1),try(Name1,Value),
                  groupname(Attr,2,Name2),try(Name2,Value),
                  groupname(Attr,3,Name3),try(Name3,Value),
                  groupname(Attr,4,Name4),try(Name4,Value),
                  groupname(Attr,5,Name5),try(Name5,Value).

/*****************************************************************/
  continue3(L,A,Value,Attr) :- filename(3,Name),tell(Name),qsort(L,Sl),
                        length(Sl,Nodist),ith(Sl,2,[Mode2,Modf2]),
                        Half is Nodist/2,round(Half,Newhalf),
                        I is Newhalf+1,ith(Sl,I,[_,Medf]),
                        writeq(data(modf2,A,Value,Modf2)),write('.'),nl,
                        writeq(data(medf,A,Value,Medf)),write('.'),nl.
  round(0.5,1).
  round(X,X) :- integer(X).
  round(X,Y) :- not(integer(X)),Y is X - 0.5.

  try(Set,Attr) :- data(freqdist,Set,Attr,L),continue(L,Set,Attr,_).
/*****************************************************************/
  continue4(L,A,Value,Attr) :- filename(4,Name),tell(Name),qsort(L,Sl),
                        puloutfreqs(Sl,L1),writeq(data(freqdist4,A,Value,L1)),

                        write('.'),nl,nl,nl.
  puloutfreqs([],[]).
  puloutfreqs([[A,B]|L1],[B|L2]) :- puloutfreqs(L1,L2).

/*****************************************************************/
  continue5(L,A,Value,Attr) :- filename(5,Name),tell(Name),
                        writeq(data(freqdist,A,Value,L)),
                        write('.'),nl,nl,nl.
```

```
/*************************************************************************/
/*************************    FIND    ************************************/
/*************************************************************************/
/*   THE PURPOSE OF THIS FILE IS TO PROVIDE MISCELLANEOUS RULES WHICH    */
/*   CONCERN FINDING A VALUE OR SOME OTHER BIT OF KNOWLEDGE ABOUT ONE OF */
/*   THE TUPLES READ IN FROM THE DATABASE.  THESE RULES ARE USED BY THE  */
/*   FREQUENCY FILE AND BY THE LEVEL 1 FILE.                             */
/*************************************************************************/
/*************************************************************************/


/*************************************************************************/
/*   THE FIND CLAUSE FINDS THE VALUE OF THE GIVEN ATTRIBUTE IN THE INPUT */
/*   SIX-TUPLE AND RETURNS THAT VALUE.  IT IS IMPORTANT TO NOTE THAT THIS*/
/*   RULE RELIES HEAVILY ON THE ASSUMPTION THAT THE DATABASE ADMINISTRA- */
/*   TOR HAS ENTERED THE ATTRIBUTES DATA ITEM IN THE ATTRLIST FILE IN    */
/*   SUCH A MANNER THAT THE ATTRIBUTE NAMES ARE LISTED IN THE SAME ORDER */
/*   THAT THE ATTIBUTE VALUES APPEAR IN THE TUPLES OF THE DATABASE.      */
/*-----------------------------------------------------------------------*/
  find(L,univ,univ).
  find(d(L),P,Attr) :- data(attributes,Attrlist),find1(L,P,Attr,Attrlist).
  find1([X|L],X,Attr,[A|Attrlist]) :- A=Attr,!.
  find1([X|L],P,Attr,[A|Attrlist]) :- find1(L,P,Attr,Attrlist).

  find(L,univ,univ).
  ith([ ],_,[0,0]).
  ith([X|L],1,X).
  ith([X|L],I,N) :- J is I-1,J>0,ith(L,J,N).

/*************************************************************************/
/*   THE FINDGROUP CLAUSE INPUTS A PARTICULAR VALUE OF AN ATTRIBUTE, THE */
/*   ATTRIBUTE ITSELF, AND THE NUMBER OF PARTITIONS THAT THE GIVEN ATTRI-*/
/*   BUTE IS DIVIDED INTO.  IT RETRIEVES THE ATTRIBUTE LIST FOR THE GIVEN*/
/*   ATTRIBUTE AND USES THE BOUNDRY VALUES STORED THERE TO CALL THE WHERE*/
/*   CLAUSE.  THE WHERE CLAUSE FINDS THE NUMBER OF THE PARTITION INTO    */
/*   WHICH THE GIVEN VALUE FALLS AND THEN THE GROUPNAME PUTS THIS NUMBER */
/*   TOGETHER WITH THE ATTRIBUTE NAME AND RETURNS A GROUPNAME INTO WHICH */
/*   THIS PARTICULAR VALUE FITS.  THE FINDSTUFF CLAUSE COMBINES THE FIND */
/*   AND THE FINDGROUP CLAUSES TO INPUT THE TUPLE AND RETURN THE GROUP   */
/*   NAME FOR THE PARTITION TO WHICH THIS TUPLE BELONGS WITH RESPECT TO   */
/*   THE INPUT ATTRIBUTE.                                                */
/*-----------------------------------------------------------------------*/
  findgroup(univ,univ,univ,_).
  findgroup(Attr,S,S,_) :- non_numeric(Attr,_).
  findgroup(Attr,S,Gpname,Attrno) :- attributelist(Attr,Attrno,L),
                                     where(S,L,Group,1),
                                     groupname(Attr,Group,Gpname).
```

```
findstuff(A,Attr,Group,_) :- A=='end_of_file',!.
findstuff(A,Attr,Group,Attrno) :- find(A,S,Attr),
                                  findgroup(Attr,S,Group,Attrno),!.

where(S,[ ],Numb,Numb).
where(S,[X|L],Numb,Numb) :- S=<X,!.
where(S,[X|L],Group,Numb) :- S>X,B is Numb+1,where(S,L,Group,B).
```

```
attributelist(temp,3,[36,37.7]).
attributelist(da,3,[15,28]).
attributelist(patno,3,[44,49]).
attributelist(pred,3,[8,20]).
attributelist(chol,3,[230,280]).
attributelist(patno,6,[10,25,44,49,63]).
attributelist(da,6,[6,10,15,20,30]).
attributelist(temp,6,[36,36.8,37,37.3,37.6]).
attributelist(chol,6,[230,250,270,304,370]).
attributelist(pred,6,[6,9,14,20,30]).
attributelist(patno,8,[6,25,40,47,49,63,78]).
attributelist(da,8,[5,10,14,18,20,30,40]).
attributelist(temp,8,[36,36.7,37,37.2,37.4,37.6,38]).
attributelist(chol,8,[230,245,260,277,300,350,400]).
attributelist(pred,8,[5,7,10,17,20,25,35]).
attributelist(patno,2,[40]).
attributelist(da,2,[13]).
attributelist(temp,2,[36.5]).
attributelist(chol,2,[230]).
attributelist(pred,2,[10]).
attributelist(patno,5,[17,27,49,63]).
attributelist(da,5,[8,12,18,25]).
attributelist(temp,5,[36,36.8,37,37.4]).
attributelist(chol,5,[230,255,282,360]).
attributelist(pred,5,[7,10,20,30]).
attributelist(patno,4,[17,40,49]).
attributelist(da,4,[10,15,25]).
attributelist(temp,4,[36,37,37.4]).
attributelist(chol,4,[230,261,320]).
attributelist(pred,4,[7,15,20]).
attributelist(sex,2,[male,female]).
attributelist(univ,1,_).
attributelist(all,0,_).
/*****************************************************************/
/*************************  ATTRLIST  ****************************/
/*****************************************************************/
/*  THE PURPOSE OF THE ABOVE FACTS IS TO PROVIDE THE BOUNDRY VALUES FOR */
/*  THE VARIOUS PARTITIONINGS OF THE DATABASE TO BUILD DATABASE AB-     */
/*  STRACTS OF VARIOUS GRANULARITIES.  THESE BOUNDRIES COULD BE DEVEL-  */
/*  OPED MANUALLY BY THE USER AND EDITTED INTO THIS FILE.  HOWEVER, THE */
/*  USUAL METHOD WOULD BE TO ALLOW THE CREATE PROGRAM TO CALCULATE THE  */
/*  BOUNDRIES FOR THE GRANULARITY DESIRED BY THE USER AND TO STORE IT   */
/*  HERE SO THAT IT CAN BE USED AGAIN BY A SUBSEQUENT INVOCATION OF THE */
/*  CREATE PROGRAM WHICH DESIRES THE SAME GRANULARITY.  THE CREATE PRO-  */
/*  GRAM APPEND THE NEW BOUNDRY LISTS ONTO THE FRONT OF THIS FILE BY    */
/*  USING UNIX SYSTEM CALLS.                                            */
```

```
/*****************************************************************/
/*****************************************************************/

/*****************************************************************/
/*  THE PURPOSE OF THE FOLLOWING FACTS IS TO COMBINE AN ATTRIBUTE NAME  */
/*  WITH A PARTICULAR PARTITION NUMBER IN ORDER TO PRODUCE A GROUPNBAME.*/
/*  THESE RULES ARE USED BY BOTH THE DATABASE ABSTRACT CREATION SYSTEM  */
/*  AND THE RULE BASED QUERY SYSTEM FOR GENERATING GROUP NAMES.         */
/*  THE LAST TWO FACTS GIVE THE ATTRIBUTE NAMES FOR THIS DATABASE AND   */
/*  WHETHER THE ATTRIBUTES ARE NON-NUMERIC.                             */
/*---------------------------------------------------------------------*/
  groupname(univ,1,univ).
  groupname(Attr,Num,Name) :- ncn_numeric(Attr,N),!,valuelist(Attr,List),
                              fromlist(Num,List,Name).
  fromlist(1,[Val|Rest],Val).
  fromlist(N,[First|Rest],Val) :- M is N-1,fromlist(M,Rest,Val).
  groupname(Attr,Num,Name) :- functor(Name,Attr,1),arg(1,Name,Num).
  data(attributes,[patno,sex,da,temp,chol,pred]).
  non_numeric(sex,2).
  valuelist(sex,[female,male]).
```

```
/*****************************************************************/
/*************************     QSORT     *************************/
/*****************************************************************/
/*  THE PURPOSE OF THIS FILE IS TO SORT A KEYED LIST IN DESCENDING ORDER*/
/*  OF THE SECOND ELEMENT IN THE TWO ELEMENT TUPLES COMPRISING THE INPUT*/
/*  LIST WITHOUT REMOVING DUPLICATE ITEMS.  THE SORT IS DONE USING THE  */
/*  QUICKSORT ALGORITHM.  THE SPLITLIST CLAUSE SPLITS THE INPUT LIST    */
/*  TWO LIST THE FIRST CONTAINING ALL ITEMS GREATER THAN THE GIVEN VALUE*/
/*  AND THE OTHER CONTAING ALL ITEMS LESS THAN OR EQUAL TO THE GIVEN    */
/*  ITEM.  THE QSORT CLAUSE IS THEN RECURSIVELY CALLED TO RETURN EACH   */
/*  COMPONENT LIST IN SORTED ORDER.  THEN THE CONCATENATE CLAUSE PUTS    */
/*  THE TWO LISTS BACK TOGETHER INCLUDING THE GIVEN ITEM                */
/*--------------------------------------------------------------------*/
  qsort([],[]).
  qsort([[A,B]|L],S1) :- splitlist(B,L,L1,L2),
                         qsort(L1,S11),qsort(L2,S12),
                         concatinate(S11,[[A,B]|S12],S1).
  splitlist(_,[],[],[]).
  splitlist(B,[[C,D]|L],[[C,D]|L1],L2) :- D>B,splitlist(B,L,L1,L2).
  splitlist(B,[[C,D]|L],L1,[[C,D]|L2]) :- D=<B,splitlist(B,L,L1,L2).
  concatinate([],L,L).
  concatinate([X|L1],L2,[X|L3]) :- concatinate(L1,L2,L3).
```

APPENDIX B

SOURCE LISTING FOR THE QUERY ESTIMATION SUBSYSTEM

```
/*****************************************************************/
/*************************     MASTER     ************************/
/*****************************************************************/
/*  THIS IS THE MASTER FUNCTION FOR A RULE-BASED DATABASE QUERY   */
/*  ESTIMATION SYSTEM DESIGNED TO CALCULATE UPPER AND LOWER BOUNDS*/
/*  AND ESTIMATIONS FOR ANSWERS TO A DATABASE QUERY USING ONLY    */
/*  TABULATED SUMMARY INFORMATION ABOUT THE DATABASE (THE DATABASE*/
/*  ABSTRACT).                                                    */
/*****************************************************************/
/*****************************************************************/

/*---------------------------------------------------------------*/
/*  THE FOLLOWING RULES MAKE UP THE USER INTERFACE PORTION OF THE */
/*  SYSTEM.  THE PROGRAM NEEDS TO KNOW THE LEVEL OF KNOWLEDGE DESIRED*/
/*  FOR THE QUERY, THE STATISTIC TO BE QUERIED, AND THE SET TO BE */
/*  QUERIED.  NOTE: THIS PROGRAM IS CURRENTLY (9/84) CONFIGURED TO */
/*  ACCEPT QUERIES ON THE "SIZE" STATISTIC ONLY BUT IS WRITTEN TO BE */
/*  EXPANDED TO INCLUDE QUERIES ON OTHER STATISTICS AT SOME TIME IN */
/*  THE FUTURE.  THE FACT 'FIRSTDONE' IS ASSERTED AFTER THE FIRST */
/*  QUERY IS MADE TO THE SYSTEM TO ALTER THE FIRST QUESTION ASKED BY */
/*  THE PROGRAM.  THE CLAUSE LEVEL(I) KEEPS TRACK OF THE CURRENT  */
/*  LEVEL OF THE KNOWLEDGE THAT THE PROGRAM IS USING AND THE CLAUSE */
/*  HIGHLEV(I) KEEPS TRACK OF THE HIGHEST LEVEL USED BY THE PROGRAM */
/*  SO FAR IN A SESSION.                                          */
 go :- not(firstdone),consult('sups'),consult('infs'),consult('ests'),
       consult('/work/tilden/crdbabs/alias'),asserta(firstdone),
       maybeconsult,ask(I),asserta(level(I)),asserta(highlev(I)),
       write('Do you want the normal, alternate or a special database '),
       write('abstract?'),nl,
       write('<help> for information'),nl,read(N_or_a),n_or_a(N_or_a,I),go1(I).
 n_or_a(normal,I) :- consult1(I).
 n_or_a(alternate,I) :- oconsult(I).
 n_or_a(special,I) :- sconsult(I).
 n_or_a(help,I) :- info2,write('Do you want normal or alternate fact files?'),
       nl,read(N_or_a),n_or_a(N_or_a,I).
 consultspecial(Bname,I,I) :- consult(Bname).
 consultspecial(Bname,I,J) :- I>J,write('Enter filename for level '),write(J),
                              nl,read(Filename),consult(Filename),
                              K is J + 1,consultspecial(Bname,I,K).
 ask(I) :- write('What level of knowledge do you want'),
       write(' to use for your queries?'),
       nl,read(I).
```

```
go :- firstdone,level(I),write('Do you wish to use a level of knowledge '),
      nl,write('different from your last query (i.e. '),
      write(I),write(' )'),nl,read(Ans),test(Ans).
go1(I) :- firstdone,write('Enter statistic to be queryed. <help> for help'),nl,

      read(Tstat),needhelp(Tstat,Stat),
      write('Enter set to be queryed. <help> for help.'),nl,
      read(Tset),needhelp2(Tset,Set),A is cputime,convert(Set,Supset,Infset),
      squash(Supset,N1set),squash(Infset,N2set),query(Stat,N1set,N2set,I),
      B is cputime - A,write('Time for this calculation -- '),write(B),
      nl,write('execution complete'),nl,nl,nl,
      write('**************************************************'),nl.
needhelp(Stat,Stat) :- not(Stat==help).
needhelp(Stat,Newstat) :- Stat==help,info3,
      write('Enter statistic to be queried.'),nl,read(Newstat).
needhelp2(Set,Set) :- not(Set==help).
needhelp2(Set,Newset) :- Set == help,info4,
      write('Enter set to be queried.'),nl,read(Newset).
test(yes) :- xx,ask(I),level(J),test2(I,J).
test2(I,J) :- highlev(H),H>=I,retract(level(J)),asserta(level(I)),go1(I).
test2(I,J) :- highlev(H),H<I,retract(level(J)),asserta(level(I)),
              retract(highlev(H)),asserta(highlev(I)),wipeout,
              write('Do you want normal,alternate, or special database '),
              write('abstract files?'),nl,read(N_or_a),n_or_a(N_or_a,I),go1(I).

test(no) :- level(I),go1(I).
/*-----------------------------------------------------------------*/
/*  THE CONVERT CLAUSE PERFORMS TWO PREPROCESSING FUNCTIONS ON THE  */
/*  SET WHICH THE USER HAS INPUT.  FIRST IT CONVERTS THE INFIX FORM */
/*  WHICH IS EASIER FOR THE USER TO TYPE INTO THE PREFIX FORM WHICH */
/*  THE PROGRAM UNDERSTANDS.  IT ALSO PROCESSES INPUTS WHICH CONTAIN*/
/*  SETS DEFINED BY VALUES OF THE ATTRIBUTE (I.E. TEMP(36,38)) AND  */
/*  FINDS A MINIMAL COVER FOR THAT SET AND A MAXIMAL UNION WHICH    */
/*  THAT SET COVERS.  THESE TWO UNIONS ARE USED AS THE ACTUAL INPUT */
/*  TO THE REST OF THE PROGRAM, THE FORMER FOR THE CALCULATION OF   */
/*  THE SUP AND THE LATTER FOR THE CALCULATION OF THE INF.          */
/*-----------------------------------------------------------------*/
convertint(Attr,L,U,Suplist,Inflist) :- partitions(Plist),
            member([Attr,Pno],Plist),
            attributelist(Attr,Pno,Alist),
            cover(Attr,L,U,1,Alist,Suplist),
            iscovered(Attr,L,U,1,Alist,Inflist).
convert(A,B,C) :- isatomset(A,B,C).
convert(A + B,or(L),or(N)) :- convertall([A,B],L,N).
convert(A * B,and(L),and(N)) :- convertall([A,B],L,N).
convert(Range,or(Sup),or(Inf)) :- functor(Range,Attr,_),
            arg(1,Range,L),arg(2,Range,U),convertint(Attr,L,U,Sup,Inf).
isatomset(A,A,A) :- data(size,A,_,_).
```

```
isatomset(A,Gpname,Gpname) :- alias(Partno,A,Gpname),groupname(Attr,_,Gpname),
                partitions(L),member([Attr,Partno],L),data(size,Gpname,_,_).
convertall([ ],[ ],[ ]).
convertall([X|L],[Y|L2],[Z|L3]) :- convert(X,Y,Z),convertall(L,L2,L3).
cover(Attr,L,U,Num,[ ],[Gpname]) :- groupname(Attr,Num,Gpname).
cover(Attr,L,U,Num,[X|Alist],Suplist) :- L>X,Numplus is Num+1,
                cover(Attr,L,U,Numplus,Alist,Suplist).
cover(Attr,L,U,Num,[X|Alist],[Gpname]) :- L=<X,U=<X,groupname(Attr,Num,Gpname).

cover(Attr,L,U,Num,[X|Alist],[Gpname|Suplist]) :- L=<X,U>X,
                groupname(Attr,Num,Gpname),Numplus is Num+1,
                coveru(Attr,U,Numplus,Alist,Suplist).
coveru(Attr,U,Num,[ ],[Gpname]) :- groupname(Attr,Num,Gpname).
coveru(Attr,U,Num,[X|Alist],[Gpname]) :- X>=U,
                groupname(Attr,Num,Gpname).
coveru(Attr,U,Num,[X|Alist],[Gpname|Suplist]) :- X<U,
                groupname(Attr,Num,Gpname),Numplus is Num+1,
                coveru(Attr,U,Numplus,Alist,Suplist).
iscovered(Attr,L,U,Num,[ ],[ ]).
iscovered(Attr,L,U,Num,[X|Alist],Inflist) :- L>X,Numplus is Num+1,
                iscovered(Attr,L,U,Numplus,Alist,Inflist).
iscovered(Attr,L,U,1,[X|Alist],[Gpname|Inflist]) :- minval(Attr,Min),
                L==Min,U>=X,groupname(Attr,1,Gpname),
                iscoveredu(Attr,U,2,Alist,Inflist).
iscovered(Attr,L,U,Num,[X|[Y|Alist]],[ ]) :- L=<X,U<Y.
iscovered(Attr,L,U,Num,[X],[ ]) :- L=<X,maxval(Attr,Max),Max>U.
iscovered(Attr,L,U,Num,[X|[Y|Alist]],[Gpname|Inflist]) :- L=<X,U>=Y,
                Numplus is Num+1,groupname(Attr,Numplus,Gpname),
                iscoveredu(Attr,U,Numplus,Alist,Inflist).
iscovered(Attr,L,U,Num,[X],[Gpname]) :- L=<X,maxval(Attr,Max),
                U>=Max,Numplus is Num + 1,groupname(Attr,Numplus,Gpname).
iscoveredu(Attr,U,Num,[X|Alist],[Gpname|Inflist]) :-
                U>=X,Numplus is Num+1,groupname(Attr,Numplus,Gpname),
                iscoveredu(Attr,U,Numplus,Alist,Inflist).
iscoveredu(Attr,U,Num,[X|Alist],[ ]) :- U<X.
iscoveredu(Attr,U,Num,[ ],[Gpname]) :- maxval(Attr,Max),U==Max,Numplus is Num+1,

                groupname(Attr,Numplus,Gpname).
iscoveredu(Attr,U,Num,[ ],[ ]) :- maxval(Attr,Max),U<Max.

/*-----------------------------------------------------------------*/
/*  THE SQUASH CLAUSE PERFORMS THE THIRD STAGE OF PREPROCESSING ON  */
/*  INPUT SET.  IT DETECTS ANY INPUTS OF THE FORM AND([AND([X,Y]),Z] */
/*  AND REPLACES THEM (SQUASHES THEM) TO THE FORM AND([X,Y,Z]).     */
/*  FOR REASONS OF PROGRAM ACCURRACY THIS TRANSFORMATION IS ABSOLUTE-*/
/*  ESSENTIAL.                                                     */
/*-----------------------------------------------------------------*/
squash(or([X]),X).
```

```
squash(or(L),or(N)) :- anyors(L,N).
squash(and(L),[]) :- member(or([]),L).
squash(and(L),and(N)) :- anyands(L,N).
squash(X,X) :- data(size,X,_,_).
anyors([],[]).
anyors([or([])|L2],N) :- anyors(L2,N).
anyors([or(L1)|L2],N) :- squash(or(L1),or(L3)),anyors(L2,M),append(L3,M,N).
anyors([X|L2],[Y|N]) :- not(X = or(_)),squash(X,Y),anyors(L2,N).
anyands([],[]).
anyands([and(L1)|L2],N) :- squash(and(L1),and(L3)),anyands(L2,M),
                           append(L3,M,N).
anyands([X|L2],[Y|N]) :- not(X = and(_)),squash(X,Y),anyands(L2,N).

/*-----------------------------------------------------------------*/
/* THE CONSULTI CLAUSE INPUTS THE LEVEL OF KNOWLEDGE DESIRED BY THE  */
/* USER AND CONSULTS THE PROPER FILES OF RULES AND FACTS.            */
/*-----------------------------------------------------------------*/
consulti(1) :- consult('/work/tilden/crdbabs/fact1').
consulti(2) :- consulti(1),consult('/work/tilden/crdbabs/fact2').
consulti(3) :- consulti(2),consult('/work/tilden/crdbabs/fact3').
consulti(4) :- consulti(3),consult('/work/tilden/crdbabs/fact4').
consulti(5) :- consulti(4),consult('/work/tilden/crdbabs/fact5'),
               consult('/work/tilden/crdbabs/freqinfo').
oconsult(1) :- consult('/work/tilden/crdbabs/nfact1').
oconsult(2) :- oconsult(1),consult('/work/tilden/crdbabs/nfact2').
oconsult(3) :- oconsult(2),consult('/work/tilden/crdbabs/nfact3').
oconsult(4) :- oconsult(3),consult('/work/tilden/crdbabs/nfact4').
oconsult(5) :- oconsult(4),consult('/work/tilden/crdbabs/nfact5'),
               consult('/work/tilden/crdbabs/freqinfo').
sconsult(1) :- consult('/work/tilden/crdbabs/three1').
sconsult(2) :- sconsult(1),consult('/work/tilden/crdbabs/three2').
sconsult(3) :- sconsult(2),consult('/work/tilden/crdbabs/three3').
sconsult(4) :- sconsult(3),consult('/work/tilden/crdbabs/three4').
sconsult(5) :- sconsult(4),consult('/work/tilden/crdbabs/three5'),
               consult('/work/tilden/crdbabs/freqinfo').
maybeconsult :- attributelist(_,_,_).
maybeconsult :- not(attributelist(_,_,_)),
                consult('/work/tilden/crdbabs/attrlist').
/*-----------------------------------------------------------------*/
/* THE QUERY CLAUSE INPUTS THE STATISTIC TO BE QUERIED AND THE SET TO*/
/* WHICH THE QUERY APPLIES FROM THE USER, CALLS THE DOQUAD CLAUSE    */
/* WHICH PERFORMS THE WORK OF THE PROGRAM AND THEN WRITES OUT THE    */
/* RESULTS.                                                          */
/*-----------------------------------------------------------------*/
query(Q,S1,S2,I) :- query(Q,S1,S2,I,_).
query(Q,S1,S2,I,F) :- doquad(Q,S1,S2,F,Sup,Inf,Est,Err,I),
                      nl,nl,nl,write('ANSWERS FOR LEVEL '),
                      write(I),nl,write('sup is '),write(Sup),nl,write(' inf is '),
```

```
                write(Inf),nl,write('est is '),write(Est),nl,
                write(' delta is '),write(Err),nl.
doquad(Q,S1,S2,F,Sup,Inf,Est,Err,I) :- dosup(Q,S1,F,Sup,I),!,
                                       doinf(Q,S2,F,Inf,I),!,
                                       others(Q,S1,S2,F,Sup,Inf,Est,Err).
others(Q,S1,S1,F,Sup,Inf,Est,Err) :- doest(Q,S1,F,Est),
                                     dodelta(Q,Sup,Inf,Err).
others(Q,S1,S2,F,Sup,Inf,Est,Err) :- not(S1==S2),doest(Q,S1,F,Est1),
                                     doest(Q,S2,F,Est2),Est is (Est1+Est2)/2,
                                     dodelta(Q,Sup,Inf,Err).
dodelta(Stat,Sup,Inf,Val) :- Val is Sup - Inf.

/*-------------------------------------------------------------------*/
/*                                                                   */
/*                RULES USED BY VARIOUS FUNCTIONS                    */
/*                                                                   */
/*-------------------------------------------------------------------*/
/*  (1)  THE MINL CLAUSE INPUTS A LIST OF INTEGERS AND RETURNS THE   */
/*       MINIMUM VALUE IN THE LIST. THE MIN CLAUSE SIMPLY RETURNS THE */
/*       MINIMUM OF TWO INTEGERS.                                    */
        min(X,Y,X) :- X=<Y.
        min(X,Y,Y) :- Y=<X.
        minl([A],A).
        minl([X|L],Y) :- minl(L,A),min(X,A,Y).

/*  (2)  THE MAX CLAUSES ARE EXACTLY ANALOGOUS TO THE MIN CLAUSES.   */
        maxl([],0).
        max(X,Y,X) :- X>=Y.
        max(X,Y,Y) :- Y>=X.
        maxl([X],X).
        maxl([X|L],A) :- maxl(L,B),max(X,B,A).

/*  (3)  THE ISDATA CLAUSE TESTS TO SEE IF THE INPUT SET HAS THE     */
/*       SPECIFIED DATA RECORDED ABOUT IT IN THE INTERNAL PROLOG     */
/*       DATA BASE. THIS KNOWLEDGE MAY BE STORED IN THE FORM OF      */
/*       DATA WHICH WAS INPUT FROM THE DATABASE ABSTRACT FILE OR      */
/*       AS A CACHE VALUE WHICH HAS PREVIOUSLY BEEN CALCULATED BY    */
/*       THE PROGRAM DURING THIS SESSION. IF THE DATA IS PRESENT IT  */
/*       IS RETURNED IN THE VAL VARIABLE, IF NOT THE CLAUSE FAILS.   */
        isdata(S,Set,E,Val) :- data(S,Set,_,Val).
        isdata(S,Set,E,Val) :- cache(S,Set,_,Val,E).
        isdata(S,Set,E,Val,Attr) :- data(S,Set,Attr,Val).
        isdata(S,Set,E,Val,Attr) :- cache(S,Set,Attr,Val,E).

/*  (4)   THE ISDISJOINT CLAUSE IS A STUB FOR THE DISJOINT CLAUSE    */
/*        WHICH JUST RETURNS THE INPUT VALUE IF DISJOINT FAILS AND   */
/*        RETURNS ZERO IF IT SUCCEDES.  THE DISJOINT CLAUSE TESTS TO */
/*        SEE IF ANY COMBINATION OF TWO OF THE SETS IN THE INPUT LIST */
```

```
/*      ARE DISJOINT. THE FIRST SET IN THE LIST IS TESTED AGAINST   */
/*      EACH REMAINING SET RECURSIVELY UNTIL THE LIST IS EXHAUSTED  */
/*      THEN THE FIRST ITEM IS DISREGARDED AND THE NEXT ITEM IS     */
/*      TESTED AGAINST THE REST OF THE LIST.  FINALLY WHEN EACH ITEM*/
/*      OF THE LIST HAS BEEN TESTED, THE LIST IS REVERSED AND THE   */
/*      PROCESS REPEATED TO ENSURE THAT NO DISJOINT SETS WERE       */
/*      MISSED.  THE S VARIABLE IS USED TO ENSURE THAT THE LIST IS  */
/*      REVERSED ONCE.                                              */
       isdisjoint(L,Input,0) :- disjoint(L,1).
       isdisjoint(L,Input,Input).
       disjoint([A],S) :- !,fail.
       disjoint(L,S) :- valuelist(_,Vlist),memberall(Vlist,L).
       memberall([],L).
       memberall([First|Rest],L) :- member(First,L),memberall(Rest,L).

      /* A CONJUNCTION OF SETS IS DISJOINT WITH THE REST OF THE */
      /* LIST IF ANY OF THE SETS OF THE CONJUNCTION IS DISJOINT */
      /* WITH THE REST OF THE LIST                              */
       disjoint([and([X])|L],S) :- disjoint([X|L],S).
       disjoint([and([X|L1])|L],S) :- disjoint([X|L],S).
       disjoint([and([X|L1])|L],S) :- disjoint([and(L1)|L],S).

      /* A DISJUNCTION OF SETS IS DISJOINT WITH THE REST OF THE */
      /* LIST IF ALL OF THE SETS IN THE DISJUNCTION ARE DISJOINT*/
      /* WITH THE REST OF THE LIST.                             */
       disjoint([or([X])|L],S) :- disjoint([X|L],S).
       disjoint([or([X|L1])|L],S) :- disjoint([X|L],S),disjoint([or(L1)|L],S).


      /* A LIST OF SETS IS DIJOINT IF IT CONTAINS TWO SETS PAR- */
      /* TITIONED ON THE SAME ATTRIBUTE BUT DIFFERENT SETS OF   */
      /* THAT PARTITION.                                        */
       disjoint([First|List],S) :- not(First=or(_)),not(First=and(_)),
                                   disjointany(First,List).
       disjointany(First,L) :- functor(First,Val,Num),functor(Look,Val,Num),
                 arg(1,First,Arg1),member(Look,L),arg(1,Look,Arg2),
                 not(Arg1=Arg2).

      /* MISCELLENANEOUS RULES */
       disjoint([X|L],S) :- disjoint(L).
       disjoint(L,1) :- reverse(L,N),disjoint(N,0).
       reverse([],[]).
       reverse([X|L],L2) :- reverse(L,L3), append(L3,[X],L2).
       append([],L,L).
       append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

/* (5)  THE TESTALLU AND TESTALLI CLAUSES TEST EACH MEMBER OF A     */
/*      LIST TO SEE IF IT IS A SUBSET OF ANY OTHER SET IN THE LIST */
```

```
/*      OR CONTAINS ANY OTHER SET IN THE LIST, RESPECTIVELY. THESE */
/*      CLAUSES RETURN THE INPUT SET MINUS ANY MEMBERS FOUND TO    */
/*      A SUBSET OF ANOTHER SET FOR TESTALLU OR FOUND TO CONTAIN   */
/*      ANOTHER SET FOR TESTALLI. (RE A+B+C+D = A+B+C IF D=<C AND  */
/*      A*B*C*D = A*B*C IF D>C).                                   */
 testallu(_,[],[]).
 testallu(L1,[X|L2],A) :- (subsetany(X,L1);subsetany(X,L2)),
                          testallu(L1,L2,A),!.
 testallu(L1,[X|L2],[X|A]) :- testallu([X|L1],L2,A).
 testalli(_,[],[]).
 testalli(L1,[X|L2],A) :- containsany(X,L1),
                          testalli(L1,L2,A),!.
 testalli(L1,[X|L2],A) :- containsany(X,L2),
                          testalli(L1,L2,A),!.
 testalli(L1,[X|L2],[X|A]) :- testalli([X|L1],L2,A).
 containsany(X,[]) :- !,fail.
 containsany(X,[Y|L]) :- subset(Y,X).
 containsany(X,[Y|L]) :- containsany(X,L).
 subsetany(X,[]) :- !,fail.
 subsetany(X,[Y|L]) :- subset(X,Y).
 subsetany(X,[Y|L]) :- subsetany(X,L).
 subset(Set1,Set1).
/* ONE CONJUNCTION IS THE SUBSET OF ANOTHER IF ALL SETS */
/* INVOLVED IN THE SECOND CONJUCTION ARE INCLUDED IN    */
/* THE FIRST.                                           */
 subset(and(L1),and([Y])) :- member(Y,L1).
 subset(and(L1),and([X|L2])) :- member(X,L2),subset(and(L1),and(L2)).

/* ONE DISJUNCTION IS THE SUBSET OF ANOTHER IF ALL SETS */
/* INVOLVED IN THE FIRST DISJUNCTION ARE INCLUDED IN    */
/* THE SECOND.                                          */
 subset(or([]),L2).
 subset(or([X|L1]),or(L2)) :- member(X,L2),subset(or(L1),L2).

/* A SINGLE SET IS THE SUBSET OF A CONJUNCTION IF IT IS  */
/* A SUBSET OF ALL SETS IN THE CONJUNCTION.              */
 subset(X,and([Y])) :- subset(X,Y).
 subset(X,and([Y|L])) :- subset(X,Y),subset(X,L).

/* A SINGLE SET IS THE SUBSET OF A DISJUNCTION IF IT IS  */
/* THE SUBSET OF ANY SETS IN THE DISJUNCTION.            */
 subset(X,or([Y])) :- subset(X,Y).
 subset(X,or([Y|L])) :- subset(X,Y).
 subset(X,or([Y|L])) :- subset(X,L).

/* A CONJUNCTION IS THE SUBSET OF A SINGLE SET IF ANY SET*/
/* IN THE CONJUNCTION IS A SUBSET OF THE SINGLE SET.     */
 subset(and(L),X) :- memberi(X,L).
```

```
memberi(X,[and(N)|L]) :- memberi(X,N).
memberi(X,[and(N)|L]) :- memberi(X,L).
memberi(X,L) :- member(X,L).

/* A DISJUNCTION IS THE SUBSET OF A SINGLE SET IF ALL    */
/* SETS IN THE DISJUNCTION ARE SUBSETS OF THE SINGLE SET.*/
subset(or(L),X) :- memberu(X,L).
memberu(X,[or(N)|L]) :- memberu(X,N).
memberu(X,[or(N)|L]) :- memberu(X,L).
memberu(X,L) :- member(X,L).

/* STANDARD TEXTBOOK MEMBER FUNCTION       */
member(X,[]) :- !,fail.
member(X,[X|L]) :- !.
member(X,[Y|L]) :- member(X,L).

/*  (6)  THE FOLLOWING CLAUSES PERFORM ARITHMETIC FUNCTIONS ON LISTS */
/* MAPDIFF INPUTS A NUMBER TO SUBTRACT FROM AND A LIST   */
/* OF INTEGERS.  IT OUTPUTS A NEW LIST WITH EACH ELE-    */
/* MENT OF THE OLD LIST SUBTRACTED FROM THE SPECIFIED    */
/* NUMBER.  THE LISTDIFF CLAUSE COMPUTES THE ELEMENT-    */
/* DIFFERENCE OF TWO LISTS. SETDIFF FINDS THE DIFFER-    */
/* ENCE OF TWO SETS IN THE MATHEMATICAL SENSE.           */
mapdiff(X,[],[]).
mapdiff(X,[Y|L1],[Z|L2]) :- Z is X-Y,mapdiff(X,L1,L2).
listdiff([],[],[]).
listdiff([A|L1],[B|L2],[C|L]) :- C is A-B,listdiff(L1,L2,L).
setdiff([],[],[]).
setdiff([X|L1],L2,L) :- member(X,L2),setdiff(L1,L2,L).
setdiff([X|L1],L2,[X|L]) :- setdiff(L1,L2,L).

/* THE SUM CLAUSE INPUTS A LIST AND OUTPUTS THE SUM OF   */
/* THE ELEMENTS OF THE LIST.  THE SUMLIST CLAUSE COM-    */
/* PUTES THE ELEMENTWISE SUM OF TWO LISTS.  THE SUMALL   */
/* CLAUSE COMPUTES THE ELEMENTWISE SUM OF ANY NUMBER OF  */
/* LISTS.  THE SUMKLIST (SUM KEYED LIST) CLAUSE INPUTS   */
/* A LIST WHICH CONTAINS TWO ELEMENT LISTS AND SUMS THE  */
/* SECOND ELEMENT OF THCSE LISTS FOR ALL THE TWO ELE-    */
/* MENT LISTS IN THE INPUT LIST.                         */
sum([],0).
sum([X|L],N) :- sum(L,S),N is S+X.
sumlist([],[],[]).
sumlist([A|L1],[B|L2],[C|L]) :-  C is A+B,sumlist(L1,L2,L).
sumall([L1,L2],L) :- sumlist(L1,L2,L).
sumall([L1|Rest],L) :- sumall(Rest,N),sumlist(N,L1,L).
sumklist([],0).
sumklist([[I,Freq]|Rest],Sum) :- sumklist(Rest,Old),Sum is Old+Freq.
```

```
/* THE PROD CLAUSE INPUTS A LIST AND OUTPUTS THE        */
/* PRODUCT OF THE ELEMENTS OF THE LIST.                 */
prod([ ],1).
prod([X|L],P) :- prod(L,A),P is A*X.

/*  (7)   THE BOTHALL, ALLFOUR, GETFREQS AND GETFREQS5 CLAUSES ALL */
/*        RETIEVE INFORMATION FRCM THE DATABASE ABSTRACT TO BE USED*/
/*        BY THE RULES FOR EACH LEVEL.  BOTHALL GETS THE MODE AND  */
/*        NUMBER OF DISTINCT ITEMS.  ALLFOUR GETS THESE TWO AND IN */
/*        ADDITION GETS THE MEDIAN FREQUENCY AND THE SECOND MOST   */
/*        COMMON FREQUENCY.  THE TWO GETFREQ CLAUSES RETRIEVE THE  */
/*        LEVEL 4 AND LEVEL 5 FREQUENCY DISTRIBUTION INFORMATION   */
bothall([ ],_,[ ],[ ],_).
bothall([F|L],E,[X|A],[Y|B],Attr) :- isdata(modefreq,F,sup,X,Attr),
                                     isdata(nodist,F,E,Y,Attr),
                                     bothall(L,E,A,B,Attr).
allfour([ ],_,[ ],[ ],[ ],[ ],_).
allfour([F|L],E,[W|A],[X|B],[Y|C],[Z|D],Attr) :-
                                     isdata(modefreq,F,sup,W,Attr),
                                     isdata(medf,F,E,X,Attr),
                                     isdata(modf2,F,E,Y,Attr),
                                     isdata(nodist,F,E,Z,Attr),
                                     allfour(L,E,A,B,C,D,Attr).
allfour([F|L],E,[W|A],[X|B],[Y|C],[Z|D],Attr) :-
                                     not(isdata(modefreq,F,sup,W,Attr)),
                                     dosup3(size,F,Attr,_,3),
                                     allfour([F|L],E,[W|A],[X|B],[Y|C],[Z|D],Attr).
getfreqs([ ],_,[ ],_).
getfreqs([X|L],E,[F|Fl],Attr) :- isdata(freqdist4,X,E,F,Attr),
                                 getfreqs(L,E,Fl,Attr).
getfreqs5([ ],_,[ ],_).
getfreqs5([X|L],E,[F|Fl],Attr) :- isdata(freqdist,X,E,F,Attr),
                                  getfreqs5(L,E,Fl,Attr).

/*  (8)   THE DOLISTS CLAUSE IS USED BY THE LEVEL 5 SUP UNION RULES */
/*        AND THE LEVEL5 INF INTERSECTION RULES TO CALCULATE THE   */
/*        SUM OF THE FREQUENCIES FOR A PARTICULAR ITEM IN ALL THE  */
/*        LISTS IN THE INPUT LIST OF LISTS.  THE QMEMBER CLAUSE    */
/*        RETURNS THE FREQUENCY FOR A GIVEN ITEM IF THE ITEM IS IN */
/*        FACT INCLUDED IN THE LIST OR RETURNS ZERO OTHERWISE.     */
dolists(_,[ ],0).
dolists(Item,[First|Rest],Sum) :- qmember(Item,First,New),
                                  dolists(Item,Rest,Old),Sum is New+Old.
qmember(Item,List,Freq) :- member([Item,Freq],List),!.
qmember(Item,List,0).

qswitch(Sofar,New,Sofar) :- Sofar =< New.
```

```
        gswitch(Sofar,New,New) :- New < Sofar.

/*   (9)   THE EXTENDLISTS CLAUSE IS USED BY THE LEVEL 4 SUP UNION    */
/*         AND LEVEL4 INF UNION RULES.  IT INPUTS A LIST OF LISTS,    */
/*         ALL OF WHICH MAY BE DIFFERENT LENGTHS, DETERMINES WHICH    */
/*         IS THE LONGEST, AND THEN ADDS AN APPOPRIATE NUMBER OF 0'S  */
/*         TO THE ENDS OF THE OTHER LISTS TO MAKE THEM AS LONG AS THE*/
/*         LONGEST LIST.  THIS WILL ENABLE THE LISTS TO BE SUMMED     */
/*         EASILY USING THE SUMLIST CLAUSE.                           */
        extendlists(L,El) :- longest(L,0,Length),extendall(L,Length,El).
        longest([ ],Len,Len).
        longest([L1|L],Old,Len) :- length(L1,New),New>Old,longest(L,New,Len).
        longest([L1|L],Old,Len) :- length(L1,New),New=<Old,longest(L,Old,Len).
        extendall([ ],_,[ ]).
        extendall([L1|L],Len,[L1|El]) :- length(L1,Len),extendall(L,Len,El).
        extendall([L1|L],Len,[N|El]) :- length(L1,S),addzeros(L1,Len,S,N),
                                        extendall(L,Len,El).
        addzeros(L1,Len,S,N) :- Add is Len-S,makzerlist(Add,L2),
                                append(L1,L2,N).
        makzerlist(1,[0]).
        makzerlist(Add,[0|L]) :- Newadd is Add-1,makzerlist(Newadd,L).

/*  (10)  THE  MAP  RULE IS A FUNCTIONAL FORM WHICH ACCEPTS AS ITS   */
/*        FIRST ARGUMENT THE NAME OF A FUNCTION TO BE EXECUTED MANY  */
/*        TIMES.  THE FUNCTION IS TO TAKE A DIFFERENT ELEMENT FROM   */
/*        THE LIST WHICH MAKES UP ITS SECOND ARGUMENT TO SERVE AS    */
/*        FIRST ARGUMENT OF THE FUNCTION CALL ON EACH EXECUTION.     */
/*        THE VALUE LIST RETURNS A LIST OF THE ANSWERS RETURNED FOR  */
/*        EACH OF THESE CALLS.  THE FUNCTION USED AS THE ARGUMENT TO*/
/*        THIS FUNCTIONAL FORM CAN HAVE ANY NUMBER OF ARGUMENTS      */
/*        BUT THE ARGUMENT TO BE VARIED MUST BE ITS FIRST ARGUMENT   */
/*        AND THE ARGUMENT WHICH RETURNS ITS ANSWER MUST BE ITS LAST*/
/*        ARGUMENT.                                                  */
        map(_,[ ],_,[ ]).
        map(Functor,[Arg1|Arglist],Otherargs,[Val|Others]) :-
                        makelist(Functor,Arg1,Otherargs,Val,Function),
                        X =.. Function,call(X),
                        map(Functor,Arglist,Otherargs,Others).
        makelist(F,A1,Otherargs,Val,Resultlist) :-
                                append([F|[A1|Otherargs]],[Val],Resultlist).
        xx :- abolish(cache,5).
        wipeout :- abolish(data,4),xx.
   :- nl,write('*************************************************'),nl,
      write('To begin program enter "go" followed by a period.'),nl,
      write('For information on the sets available for queries, '),nl,
      write('enter "info", followed by a period.'),nl,
      write('*************************************************'),nl,nl.
        info :- not(isinfo),consult('info'),info.
```

```
info2 :- not(isinfo),consult('info'),info2.
info3 :- not(isinfo),consult('info'),info3.
info4 :- not(isinfo),consult('info'),info4.
second1(L,Sec) :- max1(L,Max),setdiff(L,[Max],New1),max1(New1,Sec).
```

```
/*****************************************************************************/
/*************************        SUPS       *********************************/
/*****************************************************************************/
/*  THIS FILE CONTAINS THE RULES FOR CALCULATING THE LEAST UPPER BOUND       */
/*  OR SUPREMUM (SUP) FOR A STATISTIC.  THE RULES ARE SEPARATED INTO         */
/*  GENERAL RULES, THOSE FOR INTERSECTIONS OF SETS AND THOSE FOR UNIONS      */
/*  OF SETS.  THIS FILE IS CONSULTED BY THE MASTER FILE.                     */
/*****************************************************************************/
/*****************************************************************************/
/*                                                                           */
/*                              GENERAL RULES                                */
/*                                                                           */
/*  THESE RULES ARE USED TO DETERMINE IF THE DESIRED INFORMATION ALREADY     */
/*  EXISTS FOR A PARTICULAR QUERY.  IF IT DOES, EITHER BECAUSE IT HAS        */
/*  PREVIOUSLY BEEN CALCULATED AND CACHED OR BECAUSE THE QUERY RELATES       */
/*  TO ATOMIC SETS WHICH HAVE DATA TABULATED ABOUT THEM IN THE DATABASE      */
/*  ABSTRACT, THEN THE TABULATED VALUES ARE RETURNED.  IF NO INFORMATION     */
/*  IS ALREADY CLACULATED, THE SET BEING QUERIED MUST BE A UNION OR          */
/*  INTERSECTION OF SOME OTHER SETS.  IN THIS CASE THE SUPALL CLAUSE         */
/*  RECURSIVELY CALLS THE DOSUP CLAUSE FOR EACH ELEMENT OF THE UNION         */
/*  OR INTERSECTION.  CONSEQUENTLY LEVELS1 THRU 5 SUPS WILL BE CALCU-        */
/*  LATED FOR EACH ELEMENT OF THE UNION OR INTERSECTION LIST.                */
/*---------------------------------------------------------------------------*/
 /* 1 */ dosup(Stat,Set,Attr,Val,I) :- data(Stat,Set,Attr,Val).
 /* 2 */ dosup(Stat,Set,Attr,Val,I) :- cache(Stat,Set,Attr,Val,sup).
 /* 3 */ supall(S,[],_,[],I).
        supall(S,[X|L],_,[B|A],I) :- dosup(S,X,_,B,I),supall(S,L,_,A,I).

/*****************************************************************************/
/*                                                                           */
/*                           INTERSECTION RULES                              */
/*                                                                           */
/*  THESE RULES APPLY WHEN THE INPUT SET TO THE DOSUP CLAUSE IS AN IN-       */
/*  TERSECTION OF OTHER SETS.  THE FIRST RULE CHECKS TO SEE IF ANY OF        */
/*  THE SETS ARE DISJOINT WHICH WOULD IMPLY THAT THE SUP OF THE SIZE OF      */
/*  THEIR INTERSECTION IS ZERO.  THE FIRST RULE ALSO ASSERTS INTERMED-       */
/*  INFORMATION ABOUT THE MODE FREQUENCY, NUMBER OF DISTINCT ITEMS, ETC      */
/*  OF SUCH A DISJOINT INTERSECTION SO THAT THIS INFORMATION WILL BE         */
/*  AVAILABLE FOR THE HIGHER LEVEL RULES IF THE DISJOINT INTERSECTION        */
/*  IS A COMPONENT SET IN SOME OTHER UNION OR INTERSECTION.  THE SECOND       */
/*  RULE APPLIES WHEN THE INTERSECTION CANNOT BE DETERMINED TO BE DIS-       */
/*  JOINT.  IT CALCULATES THE VARIOUS LEVELS OF KNOWLEDGE UP TO THAT         */
/*  DESIRED BY THE USER AND OUTPUTS THE LEAST UPPER BOUND FOUND BY           */
/*  APPLYING THESE RULES.                                                    */
/*---------------------------------------------------------------------------*/
```

```
/* 1 */ dosup(size,and(L),_,0,I) :- disjoint(L,1),supall(size,L,sup,_,I),
                                    asserta(cache(modefreq,and(L),_,0,sup)),
                                    asserta(cache(nodist,and(L),_,0,sup)),
                                    asserta(cache(medf,and(L),_,0,sup)),
                                    asserta(cache(modf2,and(L),_,0,sup)),
                                    asserta(cache(freqdist4,and(L),_,[0],sup)),
                                    asserta(cache(freqdist,and(L),_,[0,0],sup)),
                                    asserta(cache(size,and(L),_,0,sup)),!.
/* 2 */ dosup(size,and(L),_,Val,I) :- supall(size,L,sup,A,I),
                                       minl(A,C),
                                    maybedo2(size,and(L),_,D,C,I),
                                    maybedo3(size,and(L),_,E,D,I),
                                    maybedo4(size,and(L),_,F,E,I),
                                    maybedo5(size,and(L),_,Val,F,I),
                                    asserta(cache(size,and(L),_,Val,sup)).

/*************************    LEVEL2 RULES   *************************/
/*  THE MAYBEDO2 CLAUSE DETERMINES IF THE USER HAS SPECIFIED A LEVEL OF */
/*  KNOWLEDGE GREATER THAN OR EQUAL TO 2.  THE DOSUP2 CLAUSE CALCULATES */
/*  THE LEVEL2 LEAST UPPER BOUND FOR THE INTERSECTION OF AN ARBITRARY   */
/*  NUMBER OF SETS GIVEN THE MODE FREQUENCY AND NUMBER OF DISTINCT ITEMS*/
/*  IN EACH OF THE SETS USING THE FOLLOWING FORMULA:                    */
/*                r      s                 s                          */
/*              min [ (min modefreq(i,j)) * min(nodist(i,j)) ]         */
/*              j=1   i=1                 i=1                          */
/*    WHERE J RANGES OVER ALL THE ATTRIBUTES AND I RANGES OVER THE SETS */
/*    IN THE INTERSECTION.                                              */
/*  THE FINDPROD CLAUSE IMPLEMENTS THE PORTION OF THE FORMULA INSIDE THE*/
/*  SQUARE BRACKETS FOR EACH OF THE ATTRIBUTES AND THE MINL CLAUSE IM-   */
/*  PLEMENTS THE OUTER MINIMUM OVER THE ATTRIBUTES.  BOTHALL MAKES A     */
/*  LIST OF THE MODEF AND NODIST FOR ALL THE SETS OF THE INTERSECTION.   */
/*  THE TWO ASSERTSION IN EACH FINDPROD CLAUSE ENSURE THAT IF THIS SET   */
/*  IS A COMPONENT OF SOME OTHER UNION OR INTERSECTIO, THERE WILL BE     */
/*  TABULATED LEVEL 2 INFORMATICN ABOUT THEM.  THESE RULES ASSUME THAT   */
/*  AN UPPER BOUND ON THE MODEF AND NODIST OF THE INTERSECTION IS THE    */
/*  MINIMUM MODEF AND NODIST OF ALL THE SETS OF THE INTERSECTION.        */
/*-----------------------------------------------------------------------*/
        maybedo2(S,and(L),_,D,D,1).
        maybedo2(S,and(L),_,0,C,I) :- I >= 2,C==0,!.
        maybedo2(S,L,_,Val,C,I) :- I >= 2,dosup2(S,L,_,D),min(C,D,Val).
        dosup2(size,and(L),_,Val) :- data(attributes,Alist),
                                     map(findprod,Alist,[L],Vlist),
                                     minl(Vlist,Val).
        findprod(Attr,L,Val) :- bothall(L,sup,A,B,Attr),
                                minl(A,X),minl(B,Y),Val is X*Y,
                                asserta(cache(modefreq,and(L),Attr,X,sup)),
                                asserta(cache(nodist,and(L),Attr,Y,sup)).
```

```
/*****************************   LEVEL3 RULES   ****************************/
/*  THE MAYBEDO3 CLAUSE DETERMINES IF THE USER DESIRES A LEVEL OF KNOW- */
/*  LEDGE GREATER THAN OR EQUAL TO 3.  THE DOSUP3 CLAUSE CALCULATES THE */
/*  LEVEL 3 LEAST UPPER BOUND FOR THE INPUT INTERSECTION OF SETS GIVEN  */
/*  THE MODE FREQUENCY, NUMBER OF DISTINCT ITEMS, SECOND MOST FREQUENT  */
/*  ITEM AND THE MEAN FREQUENCY FOR EACH OF THE COMPONENT SETS OF THE   */
/*  INTERSECTION.  IT IMPLEMENTS THE FOLLOWING 2 FORMULAS:              */
/*     r   s                  s                  s                     */
/* 1) min[ min(modefreq(i,j)) + (min modef2(i,j))*((min nodist(i,j)-1) ] */
/*    j=1 i=1                  i=1                 i=1                  */
/* AND                                                                 */
/*     r     s                  s                   s                  */
/* 2) min[ ((min modefreq(i,j) + (min meanf(i,j)))*(0.5*min nodist(i,j)) ]*/
/*    j=1    i=1                 i=1                 i=1               */
/*    WHERE AGAIN J RANGES OVER THE ATTRIBUTES AND I OVER THE SETS OF   */
/*    INTERSECTION.                                                     */
/*  THE MINIMUM OF THESE TWO VALUES IS OUTPUT AS THE LEVEL2 SUP.  THE   */
/*  ALLFOUR CLAUSE MAKES A LIST OF EACH OF THE LEVEL 3 TABULATED DATA.  */
/*  THE FINDPROD3 CLAUSE CALCULATES BOTH OF THE PRODUCTS ABOVE AND OUT- */
/*  PUTS THE MINIMUM.  AGAIN THE CLAUSE ASSERTS THE LEVEL 3 KNOWLEDGE   */
/*  ITEMS FOR THE INPUT SET IN CASE IT IS PART OF A LRGER UNION OR      */
/*  INTERSECTION.                                                       */
/*---------------------------------------------------------------------*/

        maybedo3(S,L,_,D,D,I)  :- I<3.
        maybedo3(S,L,_,0,D,I)  :- I >= 3,D==0,!.
        maybedo3(S,L,_,Val,D,I) :- I >= 3,dosup3(S,L,_,E),min(D,E,Val).
        dosup3(size,and(L),_,Val) :- data(attributes,Alist),
                                     map(findprod3,AList,[L],Vlist),
                                     minl(Vlist,Val).
        findprod3(Attr,L,Val)  :- allfour(L,sup,A,B,C,D,Attr),
                                  minl(A,M),minl(B,Medf),minl(C,M2),minl(D,Nd),
                                  T is Nd-1,Val1 is M+M2*T,
                                  T2 is Nd/2,T3 is M+Medf,Val2 is T3*T2,
                                  min(Val1,Val2,Val),
                                  asserta(cache(modefreq,and(L),Attr,M,sup)),
                                  asserta(cache(medf,and(L),Attr,Medf,sup)),
                                  asserta(cache(modf2,and(L),Attr,M2,sup)),
                                  asserta(cache(nodist,and(L),Attr,Nd,sup)).

/*************************   LEVEL4 RULES   ****************************/
/*  THE MAYBEDO4 CLAUSE DETERMINES WHETHER THE USER DESIRES A LEVEL OF  */
/*  KNOWLEDGE GREATER THAN OR EQUAL TO 4.  THE DOSUP4 CLAUSE CALCULATES */
/*  THE LEVEL 4 SUP OF THE INPUT INTERSECTION OF SETS GIVEN A LIST IN   */
/*  DESCENDING ORDER OF THE FREQUENCIES OF EACH VALUE OF EACH ATTRIBUTE */
/*  IN THE DATABASE.  THE CALCULATION IS BASED ON THE FOLLOWING FORMULA:*/
/*                  r   d(U,j)  s                                      */
/*               min[  sum  (min freq(i,j,k)) ]                        */
```

```
/*                  j=1   k=1   i=1                                    */
/*    WHERE J RANGES OVER ALL THE ATTRIBUTES OF THE DATABASE, I OVER    */
/*    ALL THE SETS OF THE INTERSECTION AND K FROM 1 TO THE NUMBER OF    */
/*    DISTINCT ITEMS IN THE UNIVERSE FOR A PARTICULAR ATTRIBUTE.        */
/*  THE DOSUMS CLAUSE SUMS THE MINIMUM FREQUENCIES FOR A PARTICULAR     */
/*  ATTRIBUTE.  THE GETFREQUENCIES CLAUSE RETURNS A LIST OF LISTS EACH  */
/*  OF WHICH CONTAINS THE FREQUENCIES OF THE VALUES OF A PARTICULAR     */
/*  ATTRIBUTE FOR EACH SET OF THE INTERSECTION.  THE FINDSUMS CLAUSE    */
/*  FINDS THE MINIMUM OF THE FIRST ELEMENT IN EACH OF THESE SUBLISTS    */
/*  AND THAT VALUE IS SUMMED WITH WITH THE PREVIOUS RUNNING SUM.  THE   */
/*  ASSERTIONS IN THE DOSUMS CLAUSE ENSURE THAT LEVEL 4 KNOWLEDGE IS    */
/*  RECORDED FOR THE INTERSECTION SET IN CASE IT IS PART OF A LARGER    */
/*  UNION OR INTERSECTION.                                              */
/*--------------------------------------------------------------------*/
      maybedo4(S,L,_,E,E,I) :- I < 4.
       maybedo4(S,L,_,0,E,I)  :- I >= 4,E==0,!.
       maybedo4(S,I,_,Val,E,I) :- I >= 4,dosup4(S,L,_,F,I),min(E,F,Val).
       dosup4(size,and(L),_,Val,I) :- data(attributes,Alist),
                                      map(dosums,Alist,[L],Vlist),
                                      minl(Vlist,Val).
       dosums(L,Attr,Val) :- getfreqs(L,sup,Fl,Attr),findsums(Fl,Val,Al),
                             asserta(cache(freqdist4,and(L),Attr,Al,sup)).
       findsums(L,S,[X|L2]) :- not(member([],L)),firstinsets(L,Newl,Firstl),
                               minl(Firstl,X),
                               findsums(Newl,Old,L2),S is Old + X.
       findsums(L,0,[]) :- member([],L).
       firstinsets([],[],[]).
       firstinsets([[X|L]|L2],[L|L3],[X|Rest]) :- firstinsets(L2,L3,Rest).

/*************************   LEVEL5 RULES   ****************************/
/*  THE MAYBEDO5 CLAUSE DETERMINES WHETHER THE USER DESIRES TO USE LEVEL */
/*  5 KNOWLEDGE FOR HIS QUERIES.  IF HE DOES, DOSUP5 CALCULATES THE SUP  */
/*  OF THE INPUT INTERSECTION SET GIVEN A TAGGED FREQUENCY DISTRIBUTION  */
/*  FOR EACH PARTITION OF THE DATABASE AND FOR EACH ATTRIBUTE.  THE      */
/*  CLAUSE IMPLEMENTS THE FOLLOWING FORMULA:                            */
/*          r   d(U,j)  s                                               */
/*        min [  sum  ( min gfreq(i,j,k))]                              */
/*        j=1   k=1   i=1                                               */
/*    WHERE J RANGES OVER THE POSSIBLE ATTRIBUTES, I OVER THE SETS OF   */
/*    THE INTERSECTION AND K FROM 1 TO THE NUMBER OF DISTINCT ITEMS IN  */
/*    THE UNIVERSE.  GFREQ HERE REPRESENTS THE FREQUENCY OF THE GLOBALLY */
/*    NUMBERED VALUE K OF ATTRIBUTE J FOR SET I OF THE INPUT INTERSEC-  */
/*    TION.                                                             */
/*  THE SUMFREQS CLAUSE IMPLEMENTS THE SUM IN THE ABOVE FORMULA AND THEN */
/*  THE MINIMUM VALUE OVER THE ATTRIBUTES IS EXTRACTED BY THE MINL      */
/*  CLAUSE.  THE GETFREQS5 AND FINDSUM5 CLAUSES PERFORM THE SAME FUNCTION*/
/*  DESCRIBED UNDER THE LEVEL 4 RULES.  THE MINFREQOFITEM RULE REPLACES */
/*  THE FIRSTINSETS AND MINL OF THE LEVEL 4 RULES BECAUSE LEVEL 5 RULES */
```

```
/*   INVOLVE A TAGGED FREQUENCY AND THE MINIMUM EXTRACTED MUST BE A MINI- */
/*   MUM FREQUENCY FOR A PARTICULAR ITEM OF THE SET.  AGAIN THE ASSERTIONS*/
/*   ARE MADE TO ENSURE THAT LEVEL 5 KNOWLEDGE WILL BE RECORDED ABOUT THE */
/*   INPUT INTERSECTION SET IN CASE IT IS PART OF A LARGER UNION OR INTER-*/
/*   SECTION.                                                             */
/*--------------------------------------------------------------------------*/
        maybedo5(S,L,_,F,F,I)  :- I < 5.
        maybedo5(S,L,_,0,F,I)  :- I = 5,F==0,!.
        maybedo5(S,L,_,Val,F,I) :- I = 5,dosup5(S,L,_,G,I),min(F,G,Val).
        dosup5(size,and(L),_,Val,I) :- data(attributes,Alist),
                                       map(sumfreqs,Alist,[L],Vlist),
                                       min(Vlist,Val).
        sumfreqs(L,Attr,Val) :- getfreqs5(L,sup,Fl,Attr),findsum5(Fl,Val,Al),
                                asserta(cache(freqdist,and(L),Attr,Al,sup)).
        findsum5([[]|Others],0,[]).
        findsum5([[Item,Freq]|Rest]|Otherlists],Sum,[[Item,Min]|Al]) :-
                                minfreqofitem(Item,Freq,Min,Otherlists),
                                findsum5([Rest|Otherlists],Old,Al),Sum is Old+Min.
        minfreqofitem(Item,Sofar,Sofar,[]).
        minfreqofitem(Item,0,0,_).
        minfreqofitem(Item,Sofar,Min,[First|Rest]) :- qmember(Item,First,New),
                                min(Sofar,New,Sofar2),
                                minfreqofitem(Item,Sofar2,Min,Rest).

/**************************************************************************/
/*                                                                        */
/*                                UNION RULES                             */
/*                                                                        */
/*   THESE RULES APPLY WHEN THE INPUT SET IS A UNION OF OTHER SETS.  THE  */
/*   RULES ACTUALLY ONLY CALCULATE A VALUE TO BE COMPARED FOR THE SMALLEST*/
/*   UPPER BOUND FOR LEVELS ONE AND FIVE.  THE RULES FOR THE OTHER LEVELS */
/*   EXIST MERELY TO ENSURE THAT LEVEL 2,3 OR 4 KNOWLEDGE IS RECORDED IN  */
/*   THE INTERNAL DATABASE FOR THE INPUT UNION SET IN CASE THIS SET IS A  */
/*   COMPONENT SET OF SOME LARGER UNION OR INTERSECTION.  THE TESTALLU RULE*/
/*   TESTS THE INPUT UNION TO ENSURE THAT NO SETS ARE SUBSETS OF OTHERS.  */
/*   IF THEY ARE THEN THE SUBSET IS OMITTED BECAUSE IT WOULD NOT CONTRIBUTE*/
/*   TO THE SIZE OF THE UNION.  THE LEVEL 1 UNION IS MERELY THE SUM OF THE*/
/*   SIZES OF THE COMPONENT SETS OR THE SIZE OF THE UNIVERSE MINUS THE    */
/*   GREATEST LOWER BOUND ON THE SIZE OF THE INTERSECTION OF THE SETS WHICH-*/
/*   EVER IS LESS.                                                        */
/*--------------------------------------------------------------------------*/
        dosup(size,or(L),_,Val,I) :- testallu([],L,N),
                                     supall(size,N,sup,A,I),
                                     sum(A,Sum),goodsup(Sum,Val1,N,I),
                                     dosup2(size,or(N),_,_,I),
                                     dosup3(size,or(N),_,_,I),
                                     dosup4(size,or(N),_,_,I),
                                     dosup5(size,or(N),_,Val2,I),min(Val1,Val2,Val),
```

```
                                asserta(cache(size,or(L),_,Val,sup)).
        goodsup(B,Val,L,I)  :- data(size,univ,nil,U),
                               doinf(size,and(L),_,Inf,I),A is B-Inf,
                               min(A,U,Val).

/*************************   LEVEL2 RULES   ******************************/
/*  THESE RULES MERELY ASSERTS THAT AN UPPER BOUND ON THE MODE FREQUENCY OF*/
/*  THE UNION IS THE SUM OF THE MODE FREQUENCIES OF THE COMPONENT SETS.    */
/*  SINCE WE DO NOT KNOW THE MODE VALUE, IT IS POSSIBLE THAT ALL THESE    */
/*  SETS HAVE THE SAME MODEFREQUENCY VALUE BUT ARE DISJOINT IN WHICH CASE  */
/*  THE MODE FREQUENCY OF THE UNION WOULD BE THE SUM OF THE MODE FREQUEN-  */
/*  CIES.  ALSO THE RULES ASSERT THAT AN UPPER BOUND ON THE NUMBER OF DIS- */
/*  ITEMS IN A SET IS THE SUM OF THE NUMBER OF DISTINCT ITEMS IN THE COM-  */
/*  PONENT SETS.  THIS WORST CASE OCCURS WHEN THE COMPONENT SETS ARE ALL   */
/*  DISJOINT BUT THERE IS NO WAY OF DETERMINING WITH CERTAINTY THAT THE    */
/*  COMPONENT SETS ARE ALL DISJOINT.                                      */
/*        THIS RULE NEEDS SOME MORE WORK BECAUSE IT SEEMS LIKELY THAT THE */
/*  THE WORST CASES MENTIONED ABOVE COULD BE TESTED FOR AND HANDLED WITH A*/
/*  SPECIAL RULE WHILE STRICTER RULES (eg. MAX OF THE MODE FREQUENCIES)    */
/*  COULD BE APPLIED TO THE OTHER CASES.  SUCH A TEST WOULD SIGNIFICANTLY  */
/*  IMPROVE THE ACCURACY OF THE RESULTS BECAUSE USER DEFINED INPUT SETS    */
/*  DESCRIBED BY VALUES OF AN ATTRIBUTE ARE TRANSLATED INTO UNIONS OF SETS */
/*  WHICH ARE TREATED AS COMPONENTS OF THE LARGER UNION OR INTERSECTION.   */
/*  THEREFORE AN IMPROVEMENT IN THE BOUNDS OF THE MODEFREQUENCY AND NUMBER */
/*  OF DISTINCT ITEMS OF SET UNIONS WOULD EFFECT A LARGE PERCENTAGE OF THE */
/*  QUERIES.                                                              */
/*-----------------------------------------------------------------------*/
        dosup2(size,or(L),_,_,1).
        dosup2(size,or(L),_,Val,I)  :- I >= 2,data(attributes,Alist),
                                       map(findprodu,Alist,[L],_).
        findprodu(Attr,L,Val)  :- bothall(L,sup,A,B,Attr),
                                  sum(A,X),sum(B,Y),
                                  asserta(cache(modefreq,or(L),Attr,X,sup)),
                                  asserta(cache(nodist,or(L),Attr,Y,sup)).

/*************************   LEVEL3 RULES   ******************************/
/*  THE ONLY PURPOSE OF THESE RULES IS TO ASSERT VALUES FOR THE SECOND MOST*/
/*  FREQUENT ITEM AND THE MEDIAN FREQUENCY ITEM OF THE UNION SET.  THE UPPER*/
/*  BOUND USED FOR THE MODEF2 IS THE SUM OF THE MODEF2'S OF THE COMPONENT  */
/*  SETS WHILE THE UPPER BOUND USED FOR THE MEDIAN FREQUENCY IS THE AVERAGE*/
/*  MEDIAN FREQUENCY OVER THE SETS.  AGAIN FOR THE REASONS MENTIONED UNDER */
/*  LEVEL 2 RULES, THIS RULE COULD USE SOME WORK.                         */
/*-----------------------------------------------------------------------*/
        dosup3(size,or(L),_,_,I)  :- I<3.
        dosup3(size,or(L),_,_,I)  :- I >= 3,data(attributes,Alist),
                                     map(findprodu3,Alist,[L],_).
```

```
findprodu3(Attr,L,Val) :- allfour(L,sup,A,B,C,D,Attr),
                          sum(A,M),sum(B,T),length(B,Len),Medf is T/Len,
                          sum(C,M2),sum(D,Nodist),
                          asserta(cache(modefreq,or(L),Attr,M,sup)),
                          asserta(cache(medf,or(L),Attr,Medf,sup)),
                          asserta(cache(modf2,or(L),Attr,M2,sup)),
                          asserta(cache(nodist,or(L),Attr,Nodist,sup)).

/*****************************   LEVEL4 RULES   *****************************/
/*  THE ONLY PURPOSE FOR THESE RULES IS TO ENSURE THAT LEVEL 4 KNOWLEDGE IS */
/*  RECORDED FOR THE INPUT UNION SET IN CASE IT IS A COMPONENT SET IN SOME  */
/*  LARGER UNION OR INTERSECTION.  THE UPPERBOUND FOR THE UNTAGGED FREQUENCY*/
/*  DISTRIBUTION USES IN THE ITH PLACE IN THE RESULT LIST THE SUM OF THE ITH*/
/*  PLACE VALUES FOR ALL THE COMPONENT LISTS.  AGAIN THIS RULE COULD USE    */
/*  SOME WORK.                                                              */
/*-------------------------------------------------------------------------*/
        dosup4(size,or(L),_,_,I) :- I<4.
        dosup4(size,or(L),_,_,I) :- I>=4,data(attributes,Alist),
                                    map(dosumsu,Alist,[L],_).
        dosumsu(L,Attr,500) :- attrinset(L,Attr),mk20list(1,Al),
                               asserta(cache(freqdist4,or(L),Attr,Al,sup)).
        attrinset([X|Rest],Attr) :- groupname(Setattr,_,X),Setattr = Attr.
        attrinset([X|Rest],Attr) :- attrinset(Rest,Attr).
        attrinset([],Attr) :- fail,!.
        mk20list(20,[500]).
        mk20list(N,[500|Rest]) :- M is N+1,mk20list(M,Rest).
        dosumsu(L,Attr,Val) :- getfreqs(L,sup,Fl,Attr),findsumsu(Fl,Val,Al),
                               asserta(cache(freqdist4,or(L),Attr,Al,sup)).
        findsumsu(L,S,L2) :- extendlists(L,El),sumall(El,L2).

/*****************************   LEVEL5 RULES   *****************************/
/*  UNLIKE THE LEVEL 2, 3, AND 4 RULES, THE LEVEL 5 RULES FOR SUP OF SET    */
/*  UNIONS IS USED TO PRODUCE A CANDIDATE TO BE COMPARED WITH THE LEVEL1     */
/*  CALCULATION TO DETERMOINE THE LEAST UPPER BOUND FOE THE SIZE OF THE      */
/*  SET BEING QUERIED.  THIS RULE INPUTS THE TAGGED FREQUENCY DISTRIBUTION   */
/*  FOR THE COMPONENT SETS OF THE UNION FOR ALL THE ATTRIBUTES OF THE DATA-  */
/*  BASE.  THE CALCULATION IS BASED ON THE FOLLOWING FORMULA:                */
/*       r  d(U,j)        s                                                  */
/*      min( sum  { min([sum   gfreq(i,j,k)],gfreq(U,j,k))})                */
/*      j=1  k=1         i=1                                                */
/*    WHERE J RANGES OVER THE ATTRIBUTES OF THE DATABASE, K RUNS FROM 1 TO   */
/*    THE NUMBER OF DISTINCT ITEMS IN THE UNIVERSE AND I RUNS FROM 1 TO THE  */
/*    NUMBER OF COMPONENT SETS IN THE UNION.  THIS RULE SIMPLY SAYS THAT     */
/*    THE NUMBER OF A PATICULAR ITEM IN THE SET UNION IS BOUNDED BY THE SUM  */
/*    OF THE NUMBER OF THAT ITEM FOR EACH COMPONENT SET OR THE NUMBER OF     */
/*    THAT ITEM IN THE UNIVERSE WHICHEVER IS SMALLER.                        */
/*  THE SUMGFREQSU CLAUSE PERFORMS THE OUTER SUM IN THE ABOVE FORMULA AND    */
/*  THEN THE MINL CLAUSE SELECTS THE MINIMUM RESULT FROM AMONG THE RESULTS   */
```

```
/*  OF ALL THE ATTRIBUTE.  THE FINDSUM5 CLAUSE IMPLEMENTS THE INNER SUM FOR */
/*  A PARTICULAR ITEM AND DETERMINES WHICH IS LESS, THAT SUM OR THE FRE-    */
/*  QUENCY OF THAT ITEM IN THE UNIVERSE.  AGAIN ASSERTIONS ARE MADE TO PRO- */
/*  VIDE LEVEL 5 KNOWLEDGE FOR THE INPUT UNION OF SETS IN CASE IT IS PART   */
/*  OF A LARGER UNION OR INTERSECTION.                                      */
/*-------------------------------------------------------------------------*/
        dosup5(size,or(L),_,Val,I) :- I < 5,data(size,univ,nil,Val).
        dosup5(size,or(L),_,Val,I) :- I = 5,data(attributes,Alist),
                                      map(sumfreqsu,Alist,[L],Vlist),
                                      minl(Vlist,Val).
        sumfreqsu(L,Attr,Val) :- getfreqs5(L,sup,Fl,Attr),
                                 data(freqdist,univ,Attr,Ul),
                                 findsum5u(Fl,Al,Ul),sumklist(Al,Val),
                                 asserta(cache(freqdist,or(L),Attr,Al,sup)).
        findsum5u(_,[],[]).
        findsum5u(Fl,[[Uitem,Rfreq]|Al],[[Uitem,Ufreq]|Urest]) :-
                                 dolists(Uitem,Fl,Sumfreqs),min(Ufreq,Sumfreqs,Rfreq),
                                 findsum5u(Fl,Al,Urest).
```

```
/*****************************************************************************/
/*************************    INFS    ****************************************/
/*****************************************************************************/
/*  THIS FILE CONTAINS THE RULES FOR CALCULATING THE GREATEST LOWER BOUND */
/*  OR INFINUM (INF) FOR A STATISTIC.  THE RULES ARE SEPARATED INTO GEN-  */
/*  ERAL RULES, THOSE FOR INTERSECTIONS OF SETS AND THOSE FOR UNIONS OF   */
/*  SETS.  THIS FILE IS CONSULTED BY THE MASTER FILE.                     */
/*****************************************************************************/
/*****************************************************************************/
/*                                                                           */
/*                              GENERAL RULES                                */
/*                                                                           */
/*  THESE RULES ARE USED TO DETERMINE IF THE INFORMATION REQUESTED  IS    */
/*  ALREADY TABULATED IN THE DATABASE ABSTRACT.  IF IT IS THE INFORMATION */
/*  IS SIMPLY RETURNED.  IF NOT, THE INFALL CLAUSE RECURSIVELY CALLS IT-  */
/*  SELF FOR EACH COMPONENT SET IN THE INPUT UNION OR INTERSECTION.  IN   */
/*  THIS WAY LEVEL 1 THRU 5 KNOWLEDGE IS TABULATED FOR EACH COMPONENT SET */
/*  IN THE INPUT UNION OR INTERSECTION.                                   */
/*---------------------------------------------------------------------------*/
 /* 1 */ doinf(size,[],_,0,_) :- asserta(cache(size,[],_,0,inf)).
 /* 2 */ doinf(Stat,Set,Attr,Val,I) :- data(Stat,Set,Attr,Val).
 /* 3 */ doinf(Stat,Set,Attr,Val,I) :- cache(Stat,Set,Attr,Val,inf).
 /* 4 */ infall(S,[],_,[],I).
       infall(S,[X|I],_,[B|A],I) :- doinf(S,X,_,B,I),infall(S,L,_,A,I).

/*****************************************************************************/
/*                                                                           */
/*                            INTERSECTION RULES                             */
/*  THE RULES FOR LEVELS 1 AND 5 ARE THE ONLY ONES WHICH ACTUALLY CONTRI-  */
/*  BUTE TO THE COMPARISON FOR THE DESIRED QUERY.  THE LEVELS 2, 3, AND 4 */
/*  RULES ONLY EXIST TO ENSURE THAT LEVEL 2, 3 AND 4 KNOWLEDGE IS ASSERTED*/
/*  INTO THE INTERNAL DATABASE FOR A GIVEN INPUT INTERSECTION IN CASE THAT*/
/*  INTERSECTION IS A COMPONENT SET IN SOME OTHER HIGHER LEVEL UNION OR    */
/*  INTERSECTION.  THE DISJOINT CLAUSE SUCCEDES IF ANY PAIR OF THE INPUT  */
/*  SETS IS DISJOINT IN WHICH CASE A LOWER BOUND ON THE SIZE OF THE ENTIRE*/
/*  INTERSECTION WOULD OF COURSE BE ZERO.  THE ASSETIONS ARE MADE IN THAT */
/*  RULE TO ENSURE THAT LEVELS 1 THRU 5 KNOWLEDGE IS TABULATED FOR THE    */
/*  DISJOINT INTERSECTION IN CASE IT IS PART OF A HIGHER LEVEL UNION OR   */
/*  INTERSECTION.  THE TESTALLI CLAUSE ELIMINATES ANY SETS WHICH CONTAIN  */
/*  ANY OTHERSETS IN THE INTERSECTION SINCE ONLY THE SMALLER SET COULD     */
/*  MAKE A CONTRIBUTION TO THE SIZE OF THE INTERSECTION.  THE LEVEL1 INF   */
/*  CALCULATION IMPLEMENTS THE FOLLOWING FORMULA:                         */
/*                                                                           */
/*                                  s                                        */
/*              max(0,size(univ) - sum (size(univ) - size (i)))              */
```

```
/*                                        i=1                                 */
/*     WHERE I RANGES OVER ALL THE SETS OF THE INTERSECTION.                   */
/*----------------------------------------------------------------------------*/
        doinf(size,and(L),_,0,I) :- disjoint(L,1),infall(size,L,inf,_,I),
                                    asserta(cache(modefreq,and(L),_,0,inf)),
                                    asserta(cache(nodist,and(L),_,0,inf)),
                                    asserta(cache(medf,and(L),_,0,inf)),
                                    asserta(cache(modf2,and(L),_,0,inf)),
                                    asserta(cache(freqdist4,and(L),_,[0],inf)),
                                    asserta(cache(freqdist,and(L),_,[],inf)),
                                    asserta(cache(size,and(L),_,0,inf)),!.
        doinf(size,and(L),_,Val,I) :- data(size,[nil,U],testalli([],L,N),
                                      infall(size,N,_,V1,I),
                                      mapdiff(U,V1,V2),sum(V2,Sum),
                                      Temp is U-Sum,max(0,Temp,Val1),
                                      doinf2(size,and(N),_,_,I),
                                      doinf3(size,and(N),_,_,I),
                                      doinf4(size,and(N),_,_,I),
                                      doinf5(size,and(N),_,Val2,I),
                                      max(Val1,Val2,Val),
                                      asserta(cache(size,and(L),_,Val,inf)).

/*****************************   LEVEL2 RULES   *****************************/
/*  THE ONLY PURPOSE FOR THESE LEVEL 2 RULES IS TO ENSURE THAT LEVEL 2  */
/*  INFORMATION IS RECORDED FOR THE INPUT INTERSECTION SET IN CASE THIS */
/*  SET IS A COMPONENT IN A HIGHER LEVEL UNION OR INTERSECTION.  THE    */
/*  FINDMAXES CLAUSE ASSUMES THAT THE GREATEST LOWER BOUND ON THE MODE  */
/*  FREQUENCY AND NUMBER OF DISTINCT ITEMS OF THE INPUT INTERSECTION IS */
/*  ZERO. THIS IS BECAUSE THERE IS NO SURE WAY TO PROVE THAT THE INPUT  */
    INTERSECTION IS NOT DISJOINT WITH ONLY LEVEL FOUR INFORMATION.      */
/*----------------------------------------------------------------------------*/
        doinf2(size,and(L),_,_,I) :- I=1;I>4.
        doinf2(size,and(L),_,_,I) :- I>1,I<5,data(attributes,Attr_list),
                                     map(findmaxesi,Attr_list,[L],_).
        findmaxesi(Attr,L,_) :- non_numeric(Attr,_),!.
        findmaxesi(Attr,L,_) :- asserta(cache(modefreq,and(L),Attr,0,inf)),
                                asserta(cache(nodist,and(L),Attr,0,inf)).

/*****************************   LEVEL3 RULES   *****************************/
/*  AGAIN THE ONLY PURPOSE FOR THESE LEVEL 3 RULES IS TO ENSURE THAT    */
/*  LEVEL 3 INFORMATION IS RECORDED FOR THE INPUT INTERSECTION SET IN   */
/*  CASE IT IS A COMPONENT OF A HIGHER LEVEL UNION OR INTERSECTION.     */
/*  A LOWER BOUND ON THE MODEF2 AND MEDIAN FREQUENCY OF THE INTERSECTION*/
/*  IS ZERO BECAUSE THE INPUT SET COULD BE DISJOINT.                    */
/*----------------------------------------------------------------------------*/
        doinf3(size,and(L),_,_,I) :- I < 3;I>4.
        doinf3(size,and(L),_,_,I) :- (I=3;I=4),data(attributes,Attr_list),
```

```
                                map(findmaxesi3,Attr_list,[L],_).
        findmaxesi3(Attr,L,_)  :- non_numeric(Attr,_),!.
        findmaxesi3(Attr,L,_)  :- allfour(L,inf,M,Me,M2,N,Attr),
                                  asserta(cache(modf2,and(L),Attr,0,inf)),
                                  asserta(cache(medf,and(L),Attr,0,inf)).

/*****************************   LEVEL4 RULES   ****************************/
/*  THE ONLY PURPOSE OF THESE RULES IS TO ENSURE THAT LEVEL 4 INFORMA-  */
/*  TION IS TABULATED ABOUT THE INPUT INTERSECTION SET IN CASE IT IS A  */
/*  COMPONENT OF A HIGHER LEVEL UNION OR INTERSECTION.  THE LOWER BOUND */
/*  FOR THE LEVEL 4 FREQUENCY LIST OF THE INPUT INTERSECTION IS A SINGLE*/
/*  ELEMENT LIST CONTAINING THE FREQUENCY OF THE ITEM WHICH OCCURS THE  */
/*  LEAST FREQUENTLY IN ANY SET OF THE INPUT INTERSECTION.              */
/*--------------------------------------------------------------------*/
        doinf4(size,and(L),_,_,I)  :- I<4;I>4.
        doinf4(size,and(L),_,_,I)  :- I = 4,data(attributes,Attr_list),
                                      map(maxesi4,Attr_list,[L],_).
        maxesi4(Attr,L,_)  :- non_numeric(Attr,_),!.
        maxesi4(Attr,L,_)  :- getfreqs(L,inf,Fl,Attr),infsumsi(Fl,Minlist),
                              minl(Minlist,Min),
                              asserta(cache(freqdist4,and(L),Attr,[Min],inf)).
        infsumsi([],[]).
        infsumsi([L1|L],[X|Al]) :- minl(L1,X),infsumsi(L,Al).


/*****************************   LEVEL5 RULES   ****************************/
/*  THESE RULES USE THE TAGGED FREQUENCY DISTRIBUTION LISTS FOR THE COM-*/
/*  PONENT SETS OF THE INPUT INTERSECTION AND CALCULATE A LOWER BOUND   */
/*  ON THE SIZE OF THE INTERSECTION USING THE FOLLOWING FORMULA:        */
/*       r   d(U,j)          s                                          */
/*     max [ sum   { max (0,[sum{gfreq(i,j,k)} - (s-1)*gfreq(U,j,k))}] */
/*     j=1   k=1          i=1                                          */
/*    WHERE J RANGES OVER THE ATTRIBUTES, I OVER THE COMPONENT SETS OF  */
/*    THE INPUT INTERSCTION, d(U,j) REPRESENTS THE NUMBER OF DISTINCT   */
/*    ITEMS IN THE UNIVERSE AND GFREQ REPRESENTS THE FREQUENCY OF OCCUR-*/
/*    RANCE OF VALUE K FOR ATTRIBUTE J IN SET I.                        */
/*  THE ASSERTIONS EXIST TO ENSURE THAT LEVEL 5 INFORMATION EXISTS FOR  */
/*  THE INPUT SET IN CASE IT IS A PART OF A HIGHER LEVEL INTERSECTION   */
/*  OR UNION.  A LOWER BOUND ON THE LEVEL 5 FREQUENCY DISTRIBUTION LIST */
/*  IS, OF COURSE THE MINIMUM FREQUENCY FOR A PARTICULAR VALUE OF ALL   */
/*  FREQUENCUIES WITH WHICH THAT ITEM OCCURS IN ANY OF THE INPUT SETS.  */
/*--------------------------------------------------------------------*/
        doinf5(size,and(L),_,0,I)  :- I < 5.
        doinf5(size,and(L),_,Val,I)  :- I = 5,data(attributes,Attr_list),
                          map(inffreqs5,Attr_list,[L],Val_list),
                          max1(Val_list,Val).
        inffreqs5(Attr,L,0)  :- non_numeric(Attr,_),!.
        inffreqs5(Attr,L,Val)  :- getfreqs5(L,inf,Fl,Attr),
```

```
                                        data(freqdist,univ,Attr,Ul),
                                        infsum5(Fl,Al,Ul),sumklist(Al,Val),
                                        asserta(cache(freqdist,and(L),Attr,Al,inf)).

        infsum5(_,[],[]).
        infsum5(Fl,[[Uitem,Rfreq]|Al],[[Uitem,Ufreq]|Urest]) :-
                                        dolists(Uitem,Fl,Sumfreqs),length(Fl,Len)
                                        Sublen is Len-1,
                                        Ifreq is Sumfreqs - Sublen*Ufreq,
                                        max(Ifreq,0,Rfreq),infsum5(Fl,Al,Urest).
/*************************************************************************/
/*                                                                       */
/*                              UNION RULES                              */
/*                                                                       */
/*  THESE RULES APPLY WHEN THE INPUT SET IS A UNION OF OTHER SETS.  THE  */
/*  LEVEL ONE RULE FIRST TESTS TO SEE IF ANY OF THE INPUT SETS ARE SUB-  */
/*  SETS OF ANY OTHERS.  SUCH SETS ARE OMITTED BECAUSE THEY WOULD NOT    */
/*  CONTRIBUTE ANYTHING TO THE SIZE OF THE UNION.  THE RULE THEN PRO-    */
/*  CEEDS TO CALCULATE THE LEVEL OF INFORMATION UP TO THAT REQUESTED BY  */
/*  THE USER.  THE MAXIMUM OF THESE RESULTS IS RETURNED AS THE ANSWER.   */
/*-----------------------------------------------------------------------*/

        doinf(size,or(L),_,Val,I) :- testallu([],L,N),
                                        infall(size,N,_,V1,I),maxl(V1,Val1),
                                        doinf2(size,or(L),_,Val2,I),
                                        doinf3(size,or(L),_,Val3,I),
                                        doinf4(size,or(L),_,Val4,I),
                                        doinf5(size,or(L),_,Val5,I),
                                        maxl([Val1,Val2,Val3,Val4,Val5],Val),
                                        asserta(cache(size,or(L),_,Val,inf)),!.


/*****************************   LEVEL2 RULES   **************************/
/*  THESE RULES CALCULATE LEVEL 2 RESULTS WHENEVER THE USER HAS SPECI-   */
/*  FIED A LEVEL OF 2, 3, Or 4 TO BE USED FOR THE QUERY. THE RULE IMPLE-*/
/*  MENTS THE FOLLOWING FORMULA:                                         */
/*    r     s                 s                                        */
/*   max [[max modef(i,j)] + [max(n(i) - modef(i,j))]] +                 */
/*   j=1   i=1               i=1                                       */
/*       s                     s                                       */
/*     [max (n(i) - modef(i,j) + max   max(0,nodist(k,j) - nodist(i,j))]*/
/*      i=1                   k=1,ki                                   */
/*      WHERE J RANGES OVER THE ATTRIBUTES, I OVER THE SETS OF THE INPUT*/
/*      INTERSECTION, n(i) REPRESENTS THE SIZE OF SET i, AND K RANGES   */
/*      OVER ALL OF THE SETS IN THE INPUT INTERSECTION EXCEPT THE SET   */
/*      SET I.                                                          */
/*  THE BOTHALL CLAUSE RETRIEVES THE MODEF AND NODIST FOR EACH OF THE   */
/*  SETS OF THE UNION.  THE SIZEALL CLAUSE RETURNS A LIST OF ALL THE    */
/*  SETS IN THE UNION.  THE LISTDIFF CLAUSE IMPLEMENTS THE SUBTRACTION  */
```

```
/*   IN THE SECOND LINE OF THE FORMULA ABOVE.  THE MAKE_ND_LIST CLAUSE   */
/*   RUNS THROUGH ALL THE SETS OF THE UNION EXCEPT THE CURRENT SET I AND */
/*   PERFORMS THE MAX(0,...) CALCULATION FROM THE SECOND LINE ABOVE.     */
/*   THE ASSERTIONS ARE MADE TO ENSURE THAT LEVEL 2 INFORMATION IS RE-   */
/*   CORDED FOR THE INPUT SET IN CASE IT IS A PART OF A HIGHER LEVEL IN- */
/*   TERSECTION OR UNION.  A LOWER BOUND ON THE MODEFREQ OF THE UNION IS */
/*   THE MAXIMUM MODEFREQ OF ANY SET IN THE UNION WHILE A LOWER BOUND ON */
/*   THE NUMBER OF DISTINCT ITEMS IN THE UNION IS THE MAXIMUM OF THE     */
/*   NUMBER OF DISTINCT ITEMS IN ANY SET OF THE UNION.                   */
/*-----------------------------------------------------------------------*/
        doinf2(size,or(L),_,0,I) :- I=1;I>4.
        doinf2(size,or(L),_,Val,I) :- I>1,I<5,data(attributes,Attr_list),
                                      map(findmaxes,Attr_list,[L],Val_list),
                                      maxl(Val_list,Val).
        findmaxes(Attr,L,0) :- non_numeric(Attr,_),!.
        findmaxes(Attr,L,Val) :- bothall(L,inf,L1,L2,Attr),sizeall(L,S),
                                 listdiff(S,L1,Ld),make_nd_list([],L2,Nd1),
                                 sumlist(Ld,Nd1,F1),maxl(L1,Maxmf),
                                 maxl(F1,Maxcorr),Val is Maxmf+Maxcorr,
                                 asserta(cache(modefreq,or(L),Attr,Maxmf,inf)),
                                 maxl(L2,Maxnodist),
                                 asserta(cache(nodist,or(L),Attr,Maxnodist,inf)).
        make_nd_list(L1,[],[]).
        make_nd_list(L1,[X|L2],[N|Nd1]) :- calcdiffs(L1,X,Diff11),
                                           calcdiffs(L2,X,Diff12),
                                           maxl(Diff11,D1),maxl(Diff12,D2),
                                           maxl([D1,D2,0],N),
                                           make_nd_list([X|L1],L2,Nd1).
        calcdiffs([],_,[]).
        calcdiffs([Y|L],X,[D|D1]) :- D is Y-X,calcdiffs(L,X,D1).

/*****************************   LEVEL3 RULES   *****************************/
/*   THESE RULES INPUT THE FREQUENCY OF THE SECOND MOST FREQUENT ITEM AND*/
/*   THE FREQUENCY OF THE MEDIAN FREQUENCY ITEM FOR EACH SET OF THE INPUT*/
/*   UNION.  THEY CALCULATE A LOWER BOUND ON THE SIZE OF THE UNION SET   */
/*   USING THE FOLLOWING FORMULA:                                        */
/*     r     s                  s                                      */
/*    max [  max modef(i,j) +  max modef2(i,j) +                         */
/*    j=1   i=1                i=1                                      */
/*           s                                                           */
/*          max [ n(i) - modef(i,j) - modef2(i,j) +                      */
/*          i=1                                                          */
/*                s                                                      */
/*               max [ medf(k,j) - modef(i,j) + nodist(k,j) - nodist(i,j) ]]*/
/*             k=1,ki                                                    */
/*     WHERE J RANGES OVER THE ATTRIBUTES, I RANGES OVER THE COMPONENT   */
/*     SETS OF THE INPUT UNION, n(i) REPRESENTS THE SIZE OF SET i, AND   */
```

```
/*    K RANGES OVER ALL OF THE COMPONENT SETS OF THE UNION EXCEPT THE   */
/*    SET I.                                                            */
/*  THE ALLFOUR CLAUSE RETRIEVES THE MODEF, MODEF2, NODIST AND MEDF     */
/*  INFORMATION FOR EACH OF THE SETS OF THE UNION.  THE SIZEALL CLAUSE  */
/*  RETURNS A LIST OF THE SIZES OF ALL THE SETS OF THE UNION.  THE      */
/*  LISTDIFF CLAUSES IMPLEMENT THE TWO SUBTRACTTION IN THE SECOND LINE  */
/*  OF THE FORMULA ABOVE. MAKE_ND_LIST3 RUNS THROUGH ALL OF THE SETS OF */
/*  THE UNION EXCEPT THE SET I AND PERFORMS THE CALCULATION IN THE THIRD*/
/*  LINE ABOVE.  THE REMAINING SUMLISTS AND MAXLS PERFORM THE OBVIOUS   */
/*  SUMS AND MAXIMUMS FROM THE FORMULA.  THE ASSERTIONS AGAIN ENSURE    */
/*  THAT LEVEL 3 INFORMATION IS RECORDED FOR THE INPUT SET IN CASE IT   */
/*  IS A PART OF A HIGHER LEVEL UNION OR INTERSECTION.  THE MEDIAN      */
/*  FREQUENCY OF THE UNION IS TAKEN TO BE THE AVERAGE OF THE MEDIAN     */
/*  FREQUENCIES OF THE COMPONENT SETS WHILE THE MODEF2 OF THE UNION IS  */
/*  TAKEN TO BE THE SECOND HIGHEST OF THE MODE FREUENCIES OF THE COMPON-*/
/*  ENT SETS.                                                         */
/*--------------------------------------------------------------------*/
doinf3(size,or(L),_,0,I) :- I < 3;I>4.
doinf3(size,or(L),_,Val,I) :- (I=3;I=4),data(attributes,Attr_list),
                    map(findmaxes3,Attr_list,[L],Val_list),
                    maxl(Val_list,Val).
findmaxes3(Attr,L,0) :- non_numeric(Attr,_),!.
findmaxes3(Attr,L,Val) :- allfour(L,inf,M,Me,M2,N,Attr),sizeall(L,S),
                    listdiff(S,M,D1),listdiff(D1,M2,D2),
                    qmake_nd_list3([ ],[ ],N,Me,Ndl),
                    sumlist(Ndl,D2,T1),
                    maxl(T1,A),maxl(M,B),maxl(M2,C),Val is A+B+C,
                    maxl(Me,E),maxl(N,Newnd),
                    asserta(cache(modefreq,or(L),Attr,B,inf)),
                    asserta(cache(modf2,or(I),Attr,C,inf)),
                    asserta(cache(medf,or(L),Attr,E,inf)),
                    asserta(cache(nodist,or(L),Attr,Newnd,inf)).
protdiv(_,0,0).
protdiv(Dividend,Divisor,Quotient) :- not(Divisor==0),
                    Quotient is Dividend/Divisor.
qmake_nd_list3(L1,L2,N,Me,Zerolist) :- anysmalllists(N),length(Me,L),
                    makzerlist(L,Zerolist).
anysmalllists([ ]) :- fail,!.
anysmalllists([Nodist|N]) :- Nodist=<3.
anysmalllists([Nodist|N]) :- anysmalllists(N).
qmake_nd_list3(L1,L2,N,Me,Ndl) :- make_nd_list3(L1,L2,N,Me,Ndl).
make_nd_list3(L1,L2,[ ],[ ],[ ]).
make_nd_list3(L1,L2,[X|N],[Y|Me],[D|Dlist]) :-
                    calcdiffs3(L1,L2,X,Y,Diff1),calcdiffs3(N,Me,X,Y,Diff2),
                    append(Diff1,Diff2,Diff),maxl(Diff,D),
                    make_nd_list3([X|L1],[Y|L2],N,Me,Dlist).
calcdiffs3([ ],[ ],_,_,[ ]).
calcdiffs3([A|L1],[B|L2],X,Y,[D|Dlist]) :- T1 is A-X,T2 is B-Y,
```

```
                                        D is T1+T2,calcdiffs3(L1,L2,X,Y,Dlist).
/*************************   LEVEL4 RULES   ***************************/
/*  THESE RULES INPUT FOR EACH SET OF THE INPUT UNION SET A LIST CON-   */
/*  SISTING OF THE FREQUENCY AT WHICH EACH ITEM OF THE SET FOR A GIVEN  */
/*  ATTRIBUTE OCCURS.  THE RULES CALCULATE A LOWER BOUND ON THE SIZE OF */
/*  THE UNION SET USING THE FOLIOWING FORMULA:                          */
/*                    r    d(U,j)    s                                  */
/*                   max [  sum   { max freq(i,j,k)} ]                  */
/*                   j=1   k=1      i=1                                */
/*    WHERE J RANGES OVER THE ATTRIBUTES, K RANGES FROM 1 TO THE NUMBER */
/*    OF DISTINCT VALUES OF THE GIVEN ATTRIBUTE IN THE UNIVERSE, AND I  */
/*    RANGES OVER THE SETS OF THE INPUT UNION.                          */
/*  THE GETFREQS CLAUSE RETRIEVES A LIST OF THE FREQUENCY LIST FOR ALL  */
/*  THE SETS IN THE INPUT UNION.  THE EXTENDLISTS CLAUSE MAKES THESE    */
/*  LISTS WHICH WERE OF VARYING LENGTH ALL HAVE A COMMON LENGTH EQUAL TO*/
/*  THE LENGTH OF THE LONGEST INPUT LIST.  THE LISTS ARE EXTENDED BY    */
/*  APPENDING ZEROS TO THE ENDS OF THE LISTS.  THE INFSUMS CLAUSE TAKES */
/*  THE FIRST ELEMENT OF EACH OF THE FREQUENCY LISTS, FINDS THE SMALLEST*/
/*  OF ALL THESE FIRST ELEMENTS, SUMS THIS VALUE ONTO THE RUNNING SUM   */
/*  AND THEN RECURSIVELY CALLS ITSELF WITH THE LISTS MINUS THEIR FIRST  */
/*  ELEMENT.  THE AASERTIONS ARE MADE TO ENSURE THAT LEVEL 4 INFORMATION*/
/*  IS RECORDED FOR THE INPUT SET IN CASE IT IS A PART OF A HIGHER LEVEL*/
/*  UNION OR INTERSECTION.                                              */
/*---------------------------------------------------------------------*/
      doinf4(size,or(L),_,0,I) :- I < 4;I>4.
      doinf4(size,or(L),_,Val,I) :- I=4,data(attributes,Attr_list),
                                    map(maxes4,Attr_list,[L],Val_list),
                                    maxl(Val_list,Val).
      maxes4(Attr,L,0) :- non_numeric(Attr,_),!.
      maxes4(Attr,L,Val) :- getfreqs(L,inf,Fl,Attr),extendlists(Fl,El),
                            infsums(El,Val,Al),
                            asserta(cache(freqdist4,or(L),Attr,Al,inf)).
      infsums([ ],0,[ ]).
      infsums(L,S,[X|Al]) :- not(member([ ],L)),firstinsets(L,Newl,Firstl),
                             maxl(Firstl,X),infsums(Newl,Old,Al),S is Old+X.
      infsums(L,0,[ ]) :- member([ ],L).

/*************************   LEVEL5 RULES   ***************************/
/*  THESE RULES INPUT TAGGED FREQUENCY INFORMATION CONSISTING OF LIST OF*/
/*  ALL THE ITEM OF A SET AND THE FREQUNCY OF OCCURRANCE OF THAT ITEM.  */
/*  THEY USE THIS INFORMATION TO CALCULATE A LOWER BOUND ON THE SIZE OF */
/*  THE UNION OF SEVERAL SETS USING THE FOLLOWING FORMULA:              */
/*                    r    d(U,j)    s                                  */
/*                   max  [  sum  { max gfreq(i,j,k)} ]                 */
/*                   j=1     k=1    i=1                                */
/*    WHERE J RANGES OVER THE ATTRIBUTES, K RANGES FROM 1 TO THE NUMBER */
/*    OF DISTINCT TIEMS IN THE UNIVERSE, AND I RANGES OVER THE SETS OF  */
/*    THE INPUT UNION.                                                  */
```

```
/*   THE GETFREQS5 CLAUSE RETRIEVES THE LEVEL 5 INFORMATION FOR ALL OF   */
/*   THE SETS IN THE UNION.  THE INFSUM5U CLAUSE INPUTS A PARTICULAR ITEM*/
/*   FROM THOSE WHICH OCCUR IN THE UNIVERSE AND FINDS THE MAXIMUM FRE-   */
/*   QUENCY FOR THAT ITEM IN ANY OF THE SETS IN THE UNION AND SUMS THAT  */
/*   AMOUNT ONTO A RUNNING TOTLA.  THEN IT RECURSIVELY CALLS ITSELF WITH */
/*   THE NEXT ITEM IN THE  UNIVERSE AS ITS INPUT ARGUMENT.  THE ASSER-   */
/*   TIONS ENSURE THAT LEVEL 5 INFORMATION IS RECORDED FOR THE INPUT SET */
/*   IN CASE IT IS A PART OF A HIGHER LEVEL UNION OR INTERSECTION.  A    */
/*   LOWER BOUND ON THE FREQUENCY OF A PARTICULAR VALUE IN THE UNION IS  */
/*   THE MAXIMUM FREQUENCY OF OCCURRANCE OF THAT ITEM IN ANY SET OF THE  */
/*   UNION.                                                              */
/*-----------------------------------------------------------------------*/
        doinf5(size,or(L),_,0,I) :- I < 5.
        doinf5(size,or(L),_,Val,I) :- I = 5,data(attributes,Attr_list),
                                      map(inffreqs5u,Attr_list,[L],Val_list),
                                      max1(Val_list,Val).
        inffreqs5u(Attr,L,0) :- non_numeric(Attr,_),!.
        inffreqs5u(Attr,L,Val) :- getfreqs5(L,inf,Fl,Attr),infsum5u(Fl,Val,Al),

                                      asserta(cache(freqdist,or(L),Attr,Al,inf)).
        infsum5u([ ],0,[ ]).
        infsum5u([[ ]|Others],0,[ ]).
        infsum5u([[Item,Freq]|Rest]|Otherlists],Sum,[[Item,Max]|Al]) :-
                        maxfreqofitem(Item,Freq,Max,Otherlists),
                        infsum5u([Rest|Otherlists],Old,Al),Sum is Old + Max.
        maxfreqofitem(Item,Sofar,Sofar,[ ]).
        maxfreqofitem(Item,Sofar,Max,[First|Rest]) :- qmember(Item,First,New),
                        max(Sofar,New,Sofar2),
                        maxfreqofitem(Item,Sofar2,Max,Rest).
        sizeall([ ],[ ]).
        sizeall([X|L],[S|Sl]) :- isdata(size,X,inf,S),sizeall(L,Sl).
        second1(L,Sec) :- max1(L,Max),setdiff(L,[Max],New1),max1(New1,Sec).
```

```
/*************************************************************************/
/*************************    ESTS    ************************************/
/*************************************************************************/
/*   THIS FILE CONTAINS THE RULES FOR CALCULATING THE ESTIMATE FOR THE    */
/*   VALUE OF A STATISTIC.  THE RULES ARE SEPARATES INTO GENERAL RULES,   */
/*   THOSE FOR UNIONS AND THOSE FOR INTERSECTIONS.                        */
/*************************************************************************/
/*************************************************************************/
/*                                                                       */
/*                              GENERAL RULES                            */
/*                                                                       */
/*   THESE RULES ARE USED TO DETERMINE IF THE DESIRED INFORMATION HAS    */
/*   ALREADY BEEN TABULATED IN THE DATABASE ABSTRACT.  IF SO THE INFORMA- */
/*   TION IS SIMPLY RETRIEVED, IF NOT THEN THEN CALCULATIONS ARE STARTED  */
/*   TO DETERMINE A GOOD VALUE FOR AN ESTIMATE.                          */
/*-----------------------------------------------------------------------*/
 /* 1 */ doest(size,[],_,0).
 /* 2 */ doest(Stat,Set,Attr,Val) :- data(Stat,Set,Attr,Val).
 /* 3 */ doest(Stat,Set,Attr,Val) :- cache(Stat,Set,Attr,Val,est).


/*************************************************************************/
/*                                                                       */
/*                          INTERSECTION RULES                           */
/*                                                                       */
/*   THESE RULES APPLY WHEN THE INPUT SET TO THE DOEST CLAUSE IS AN INTER-*/
/*   SECTION OF ANY NUMBER OF SETS.  THE FIRST RULE CHECKS TO SEE IF ANY  */
/*   OF THE SETS CONTAINS ANY OTHER.  IF SO THE LARGER SET IS OMMITTED    */
/*   FROM THE INTERSECTION BECAUSE IT WOULD NOT CONTRIBUTE TO THE SIZE    */
/*   OF THE INTERSECTION SET.  NEXT, THE ESTALL CLAUSE DETERMINES THE     */
/*   SIZE OF EACH SET IN THE INTERSECTION BY RECURSIVELY CALLING THE DOEST*/
/*   CLAUSE FOR EACH OF THE SETS.  FINALLY THESE SIZES ARE USED TO ESTI-  */
/*   MATE THE SIZE OF THE INTERSECTION USING THE FOLLOWING FORMULA:       */
/*                    s                                                  */
/*                 [prod                                                 */
/*   (size(i))]/(U  i=1)                                                 */
/*                                                                       */
/*     WHERE I RANGES OVER ALL THE SETS IN THE INTERSECTION AND U IS THE  */
/*     SIZE OF THE UNIVERSE.                                             */
/*   A FINAL TEST IS MADE TO ENSURE THAT NONE OF THE INTERSECTION SETS    */
/*   ARE DISJOINT.  IF THEY ARE, THEN ZERO IS SUBSTITUTED FOR THE CALCU-  */
/*   LATED VALUE.  IN EITHER CASE THE RESULT IS ASSERTED INTO THE INTERNAL*/
/*   CACHE TO BE USED IF THIS INTERSECTION IS A COMPONENT IN SOME HIGHER   */
/*   LEVEL UNION OR INTERSECTION.                                        */
/*-----------------------------------------------------------------------*/
```

```
 /* 1 */ doest(size,and(L),_,Val) :- data(size,univ,nil,U),testalli([ ],L,N),
                                     estall(size,N,_,V1),
                                     prod(V1,P),length(V1,F),E is F-1,D is
U                                 E,
                                     Input is P/D,
                                     isdisjoint(L,Input,Val),
                                     asserta(cache(size,and(L),_,Val,est)).

         estall(S,[ ],_,[ ]).
         estall(S,[X|I],_,[B|A]) :- doest(S,X,_,B),estall(S,L,_,A).
         prod([ ],1).
         prod([X|L],P) :- prod(L,A),P is A*X.

/*****************************************************************************/
/*                                                                           */
/*                                UNION RULES                                */
/*                                                                           */
/*   THESE RULES APPLY WHENEVER TH INPUT SET IS A UNION OF OTHER SETS.       */
/*   FIRST THE UNION SETS ARE CHECKED TO DETERMINE IF ANY ARE SUBSETS OF     */
/*   OTHERS.  IF SO, THEN THE SMALLER SET IS OMMITTED FROM THE UNION         */
/*   SINCE ITS SIZE WOULD NOT CONTRIBUTE ANYTHING TO THE SIZE F THE          */
/*   UNION.  NEXT THE ESTALLU CLAUSE DETERMINES THE SIZE OF EACH OF THE      */
/*   SETS OF THE UNION EITHER BY RETRIEVING TABULATED VALUES OR BY PER       */
/*   FORMING THE FOLLOWING THE FCLLOWING CALCULATION ON EACH PAIR OF SETS:   */
/*                                                                           */
/*            size(or(X,Y)) = size(X) + size(Y) - size(and(X,Y))             */
/*   THE ESTALLU USES THE FIRST TWO SETS OF THE UNION AS X AND Y.  THEN      */
/*   THE ESTALLU1 CLAUSE UASES THE PREVIOUS UNION RESULT AS X AND THE NEXT   */
/*   SET OF THE UNION AS Y.  SPECIAL CASES OF EMPTY AND SINGLETON SETS       */
/*   ARE HANDLED BY SPECIAL RULES.                                           */
/*---------------------------------------------------------------------------*/
 /* 1 */ doest(size,or(L),_,Val) :- testallu([ ],L,N),
                                    estallu(N,Val),asserta(cache(size,or(L),_,Val,est)).
         estallu([X|[Y|L]],Val) :- doest(size,X,_,V1),doest(size,Y,_,V2),
                                   doest(size,and([X,Y]),_,I),S is V1+V2-I,
                                   asserta(cache(size,or([Y,X]),_,S,est)),
                                   estallu1([or([Y,X])|L],Val).
         estallu([X],Val) :- doest(size,X,_,Val),
                             asserta(cache(size,or([X]),_,Val,est)).
         estallu([ ],0) :- asserta(cache(size,or([ ]),_,0,est)).
         estallu1([or(X)|[ ]],Val) :- cache(size,or(X),_,Val,est).
         estallu1([or(X)|[Y|L]],Val) :- doest(size,or(X),_,V1),
                                        doest(size,Y,_,V2),
                                        doest(size,and([or(X),Y]),_,I),S is V1+V2-I,
                                        asserta(cache(size,or([Y|X]),_,S,est)),
                                        estallu1([or([Y|X])|L],Val).
```

```
/*************************************************************************/
/*************************    INFO    ************************************/
/*************************************************************************/
/*  THIS FILE PROVIDES THE HELP FEATURE FOR THE MASTER FILE.  IT IS NOT   */
/*  CONSULTED UNTIL THE USER REQUESTS HELP AT WHICH TIME THE FIRST CLAUSE */
/*  ISINFO TELLS THE SYSTEM THAT NOW HELP IS AVAILABLE AND THIS FILE NEED */
/*  NOT BE CONSULTED AGAIN.  THE INFO CLAUSE IS A CATCH HELP REQUEST WHICH*/
/*  PROVIDES THE USER WITH A TOTAL EXPLANATION OF THE SYSTEM.             */
/*-----------------------------------------------------------------------*/
 isinfo.
 info :- maybeconsult,write('This system requires the following input: '),nl,
         write('1) level of knowledge-- a number from 1 to 5'),nl,
         write('2) a choice of normal or alternate data base abstracts '),nl,
         info2,info3,info4.
/*************************************************************************/
/*  THE INFO2 CLAUSE PROVIDES INFORMATION ABOUT THE CURRENT PARTITIONING  */
/*  OF THE DATABASE TO FORM THE DATABASE ABSTRACT.  THE DISPLAY SETS      */
/*  CLAUSE WILL DISPLAY THE CURRENT SETTINGS FOR THE RANGE OF THE INPUT   */
/*  ATTRIBUTE'S PARTITIONS AND ANY ALIASES WHICH ARE CURRENTLY ASSOCIATED */
/*  WITH PARTICULAR PARTITION SETS.                                       */
/*-----------------------------------------------------------------------*/
 info2 :- write('Currently the normal database abstract is partitioned '),
         write('as follows '),nl,consulti(1),partitions(L),display_sets(L),
         abolish(data,4),abolish(partitions,1),
         write('       The alternate database is partitioned as follows '),
         oconsult(1),nl,partiticns(N),display_sets(N),abolish(data,4),
         abolish(partitions,1),
         write('       The special database is partitioned as follows '),
         sconsult(1),nl,partiticns(M),display_sets(M),abolish(data,4),
         abolish(partitions,1).
  display_sets([]).
  display_sets([[Attr,Pno]|Rest]) :- attributelist(Attr,Pno,Alist),
         display(Attr,Pno,1,Alist,min),display_sets(Rest).
  display(sex,2,1,_,_) :- groupname(sex,2,Name),write(Name),nl,
         groupname(sex,1,Name2),write(Name2),nl.
  display(Attr,Pno,Pno,_,Last) :- groupname(Attr,Pno,Name),write(Name),
         write(' -- greater than '),write(Last),
         write(' and less than maximum value in database'),nl,galias(Pno,Name).
  display(Attr,Pno,Num,[X|Rest],Last) :- groupname(Attr,Num,Name),write(Name),
         write(' -- greater than '),write(Last),write(' and less than '),
         write(X),nl,galias(Pno,Name),Newnum is Num + 1,
         display(Attr,Pno,Newnum,Rest,X).
  galias(Pno,Name) :- alias(Pno,Alias,Name),!,write('Alias for '),write(Name),
           write(' --> '), write(Alias),nl.
  galias(Pno,Name).
/*************************************************************************/
```

```
/*  THE INFO3 CLAUSE PROVIDES THE USER WITH INFORMATION ABOUT THE STATIS-  */
/*  TICS WHICH ARE CURRENTLY AVAILABLE FOR QUERY.                          */
/*-------------------------------------------------------------------------*/
 info3 :- write('3) the statistic to be queried.  As of this date the '),
          nl,write('   only statistic that the system is capable of processing'),

          nl,write('   is size.'),nl.
/*************************************************************************/
/*   THE INFO4 CLAUSE PROVIDES THE USER WITH INFORMATION ABOUT THE FORMAT  */
/*   OF THE INPUT SET REQUIRED BY THE SYSTEM.                              */
/*-------------------------------------------------------------------------*/
 info4 :- write('4) the set to be queried in infix form. '),nl,
          write('   eg.  The set of all patients with patient number from '),nl,
          write('   5 to 30 who are also female ---> patno(5,30) * female.'),nl.
/*************************************************************************/
/*  THIS CLAUSE ENSURES THAT THE ATTRLIST FILE IS ONLY CONSULTED ONCE BY   */
/*  CHECKING TO SEE IF IT IS ALREADY CONSULTED EACH TIME.                  */
/*-------------------------------------------------------------------------*/
  maybeconsult :- attributelist(_,_,_).
  maybeconsult :- not(attributelist(_,_,_)),
                  consult('/work/tilden/crdbabs/attrlist')
                  consult('/work/tilden/crdbabs/alias').
```

LIST OF REFERENCES

1. Rowe, N. C., Rule-Based Statistical Calculations on a Database Abstract Ph.D. Thesis, Stanford University, 1983.

2. Rowe, N. C., Absolute Bounds on Set Intersections and Union Sizes from Distribution Information technical paper prepared at Naval Postgraduate School, 1984.

BIBLIOGRAPHY

Kowalski, R. A., "Logic as A Programming Language", Logic Programming, London, England, Academic Press, 1982.

Lefons, Ezio and Silvestri, Alberto and Tangorra, Filippo, An Analytic Approach to Statistical Databases, presented at the Ninth International Conference on Very Large Databases, Florence, Italy, 1983.

Nusson, Nils J. Probablistic Logic presented at the Office of Naval Research, Menlo Park, California, 1983.

Pereira, L. M. and Porto, A., "Selective Backtracking," Logic Programming, London, England, Academic Press, 1982.

Piatetsky-Sha ro, Gregory and Connel, Charles, "Accurate Estimation of he Number of Tuples Satisfying a Condition", Sigmod Record olume 14, Number 2, Boston, Mass, June 1984.

Rowe, N. C. Absolute Bounds on the Mean and Standard Deviation of Transformed Data for Constant-Derivative Transformations technical paper prepared at Naval Postgraduate School, Monterey, California, 1983.

Rowe, N. C., Optimal Top-Down Statistical Estimation, technical paper prepared at Naval Postgraduate School, Monterey, California, 1983.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0412 Naval Postgraduate School Monterey, California 93943	2
3.	Lieutenant Barry M. Tilden c/o Frank W. Tilden 202 Greenfield Place Bristol, Tennessee 37620	4

213168

Thesis
T477
c.1

Tilden
A hierarchy of knowledge levels implemented in rule-based production system to calculate bounds on the size of intersection and unions of simple sets.

Thesis
T477
c.1

213168

Tilden
A hierarchy of knowledge levels implemented in rule-based production system to calculate bounds on the size of intersection and unions of simple sets.

A hierarchy of knowledge levels implemen
3 2768 000 61465 5
DUDLEY KNOX LIBRARY